Interactive Editing Systems: Part II

NORMAN MEYROWITZ AND ANDRIES VAN DAM

Department of Computer Science, Brown University, Providence, Rhode Island 02912

This article, Part II of a two-part series, surveys the state of the art of computer-based *interactive editing systems*. This paper is a survey intended for a varied audience, including the more experienced user and the editor-designer as well as the curious novice. It presents numerous examples of systems in both the academic and commercial arenas, covering line editors, screen editors, interactive editor/formatters, structure editors, syntax-directed editors, and commercial word-processing editors. We discuss pertinent issues in the field, and conclude with some observations about the future of interactive editing. The references for both parts are provided at the end of Part II.

Categories and Subject Descriptors: D 2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*, D.2.3 [**Software Engineering**]: Coding—*prettyprinters*; *program editors*; H.4.1 [**Information Systems Applications**]: Office Automation—*equipment*; *word processing*; I.7.0 [**Text Processing**]: General; I.7.1 [**Text Processing**]: Text Editing—*languages*; *spelling*; I.7.2 [**Text Processing**]: Document Preparation—*format and notation*; *languages*; *photocomposition*; I.7.m [**Text Processing**]: Miscellaneous

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Syntax-directed editors, structure editors

INTRODUCTION

Part I of this series (pp. 321–352) provides a reasonably comprehensive introduction to text editing, presenting definitions and an overview of the field.

Part II presents technical details of specific editors, using the terminology and concepts laid out in Part I. It is intended for a broader audience, including those quite familiar with the concepts covered in the first half as well as those comfortable with the editors in their own computing environments but not necessarily familiar with the range of editors available. This part surveys editors available in the academic and commercial realms, providing points of departure for further investigation rather than an exhaustive point-by-point comparison. We discuss unresolved issues in the field, and examine the future of editing. The reference list and bibliography at the end of Part II provide material for further reading.

1. IMPLEMENTATIONS

This survey discusses a wide variety of editors used in academic and commercial circles. Our purpose is not to provide a detailed point-by-point comparison; our coverage from editor to editor is not necessarily consistent in either subject matter or depth. Rather, using the terminology of our tutorial, "Interactive Editing Systems: Part I," (pp. 321–352) we attempt to illustrate the capabilities outlined in Part I, Section 3, of that tutorial by briefly describing the distinctive features of each editor or class of editors. While a taxonomy of the interactive editor—one in which we could compare the genealogy, purposes, and features of various systems—would be useful, it is difficult to construct. Terminology for categorizing editors is far from standard, a fact that often leads to identical labels for less than identical software and hardware. The history of editing contains many parallel develop-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0010-4892/82/0900-0353 $00.75

CONTENTS

INTRODUCTION
1. IMPLEMENTATIONS
1.1 Line-Oriented Editors
1.2 Stream Editors
1 3 Display Editors
1.4 Graphics-Based Interactive Editor/Formatters
1 5 General-Purpose Structure Editors
1 6 Syntax-Directed Editors
1.7 Word Processors
1.8 Integrated Environments
2. ISSUES
2.1 The State of Editor Design
2 2 The Modeless Environment
2 3 Instant Editor/Formatters versus Batch Formatters
2 4 Structure/Syntax-Directed Editors versus "Normal" Editors
3 CONCLUSION
3 1 Desiderata for Today's Editor
3 2 Standardization
3 3 The On-Line Community
POSTSCRIPT
ACKNOWLEDGMENTS
REFERENCES
BIBLIOGRAPHY

ments and much cross-fertilization of ideas; a strict ordering or categorization is thus impossible. Informally, then, we shall be looking at editors from the viewpoint of the *target applications*[1] for which they were designed, the *elements* and their *operations*, the nature of the *interface*, and the system *configuration*. These categories do not form strictly independent axes; the choice of one frequently influences the choice of another.

"Target applications" are the high-level entities that the editor manipulates, for example, manuscripts, programs, or pictures. "Elements" are the units of target data that may be manipulated by the user. For example, a user may manipulate a program in the units of single lines of text, of individual programming language constructs, or of individual nodes in a parse tree. User "operations" fall into several subcategories. Editing operations allow the user to manipulate the target elements. Traveling operations allow the user to browse through a document. Viewing operations allow the user to control what subset of target data is presented to the user and how it is formatted; for example, text may be viewed as single lines, as full-screen pages, as a prettyprinted program, or as a facsimile of a typeset document. "Interface" defines the interaction language, input devices, and output devices with which the user performs these operations. "Configuration" describes the architecture of the systems on which the editor can run.

For compatibility with popular terminology, we review some of the most common terms that are used in this section. A *text editor* is one of the basic components of a *text-processing system*, which is concerned not only with creation and maintenance but also with formatting and interactive presentation of text. In addition to a text editor, a text-processing system includes a *text formatter*, concerned with the layout and typography of the text, and various *text utilities* such as spelling correctors that aid in analyzing and preparing the text. *Word processing* is a commercial synonym for text processing. An *office automation system* typically combines a word-processing system with utilities such as database management, information retrieval, electronic mail, and calendar management. *Program editors* operate on programs, whether represented in textual form or in another canonical form, such as a parse tree or an abstract syntax tree. *Picture* or *graphics editors* facilitate the creation and revision of computer-based graphics. A new development in the text-processing field is the *document preparation system*, which integrates text editing, picture editing, and formatting. A *voice editor* is a specialized interactive editor in which the target is digitally encoded voice. A *forms editor* is an interactive editor that allows users to create and to fill in business forms conveniently. An *interactive editor/formatter*, often called a "what-you-see-is-what-you-get" editor/formatter, allows the user to edit a facsimile of the printed page such that the changed text is reformatted instantaneously. On standard alphanumeric terminals, the facsimile represents a monospaced, typewriter page. On high-resolution raster

[1] In this paper, italic type is used to introduce concepts and terms. Sans serif type is used to set off editor commands Boldface type is used for emphasis.

graphics displays, the facsimile represents a proportionally spaced, typeset page, with a variety of typefaces, sizes, and weights, and such nontextual material as equations, line drawings, and even photographs. The goal of the *universal* or *virtual editor*, a current topic of research, is to generalize and integrate previously target-dependent software, providing a uniform way to manipulate seemingly dissimilar targets such as manuscript text, program text, pictures, and digitized voice. A *structure editor* is a special type of virtual editor that gains its generality by imposing the same structure on different targets. For example, a structure editor based on hierarchy may allow the user to impose this tree structure on diverse targets and edit them with the same tree-editing primitives (e.g., delete subtree, move current subtree up 1 level, display all siblings of node). A *syntax-directed editor* is based on the same principles as a structure editor but imposes the syntactic structure of a particular language, rather than a general-purpose structure, on the target.

1.1 Line-Oriented Editors

Line-oriented editors are covered here simply to round out our treatment of common editors. We do not, however, advocate the continued production or use of these editors. The conceptual model presented with line editors is that of editing virtual card images; the line editor constantly visits the limitations of this outdated representation of data on the user. Notable drawbacks are pattern searches and edits that do not cross line boundaries, and overflow and subsequent truncation of fixed-length lines. The continued dependence on the card analogy illustrates an important design flaw in many editors: they adhere to outmoded conventions even though those conventions unnecessarily limit the technology of the day. Unlike the TECO stream model below, where lines are simply an optional filtering presented to the user as a service, line editors force this limited view on the user.

1.1.1 IBM's CMS Editor

IBM's CMS editor (ca. 1967) is a classic example of a fixed-length line-oriented editor with a textual interface, designed for a time-sharing system in which terminals lack cursor motion keys and function keys. It presents the user with a one-line editing buffer (the amount of the document that can be edited at a given time), although this is extended for some operations. Similarly, it presents a corresponding one-line viewing buffer (the amount of the document that is used to construct the display). The display is a simple mapping of the one-line viewing buffer to a one-line window; it is typically updated after the execution of each command. (A more thorough explanation of the editing buffer, viewing buffer, and window model is presented in Part I, Section 1.) Traveling is done with line granularity, using absolute and relative gotos to varying internal line numbers and using context pattern matching. The input language is textual with two major modes: input and edit mode. Typically the user spends most time in edit mode, with input mode reserved for bulk input of text. The prefix syntax is generally consistent across commands:

command/scope/optional destination/optional parameters

The commands are full English words; the user does not have to remember abbreviations, although the system will accept the smallest possible unambiguous abbreviation. Most commands operate on the line units, and within lines as well, if so specified by the scope.

We now show some simple editing using the CMS editor. Assume that we are in edit mode and that the following section of a program, which computes the sum of two matrices, is to be modified to compute the difference of the two matrices:

```
ADD: PROCEDURE;
FOR ROW = 1 TO N DO;
  FOR COLUMN = 1 TO M DO;
    C(ROW, COLUMN) = A(ROW, COLUMN) + B(ROW, COLUMN);
  END;
END;
```

The following sequence of interactions with the editor would provide the necessary changes (the user's requests are preceded by the system prompt character ">"):

```
>find/add:
ADD: PROCEDURE;
>change/add/subtract
SUBTRACT: PROCEDURE;
>next 3
     C(ROW, COLUMN) = A(ROW, COLUMN) + B(ROW, COLUMN);
>change/+/-
     C(ROW, COLUMN) = A(ROW, COLUMN) - B(ROW, COLUMN);
>top
>change/add/subtract/ * *
```

The routine ADD is first located by using the column-dependent find command that searches for the string "ADD:" beginning in the first position of a line. (The locate commands, after searching for a pattern that does not exist, travel to and leave the buffer either at the beginning or end of the file, frustrating the user who has erroneously specified a search pattern and must manually grope back to the former location.) The current line pointer now points to the line "ADD: PROCEDURE"; this line is echoed on the screen. The next user command, change/add/subtract, affects only the contents of the buffer: the first occurrence of "ADD" is replaced by "SUBTRACT." For appropriate types of files, the editor does automatic lowercase to uppercase translation. If the maximum line length of 132 characters is exceeded, the editor will truncate the line. Line numbers in the CMS editor are varying. Travel is specified relatively with next (next 3 moves the current pointer and hence the editing and viewing buffers three lines down from the current location), or absolutely with goto (goto 276 moves the current pointer to the 276th line of the document). The change/+/− command changes the "+" to a "−". The top command moves the current line pointer to the first line of the file. The "* *" operand of the final change specifies the replacement of **all** occurrences of "ADD" in **all** lines—this is a global change that will affect the entire file.

The CMS editor provides the ability to set up logical tab stops—tabs implemented in the editor software rather than in terminal hardware—so that tabs may be specified by typing a user-chosen logical tab character in the input stream. Certain installation-specific enhancements of the basic CMS editor allow the user to undo the most recent command, shorten the scope specification by using ellipses (. . .), and do automatic indentation tailored to language-dependent needs [BROW81].

One of the most confusing attributes of the CMS editor are its two modes. Edit mode gives the user access to all the functional capabilities of the editor, including the capability to switch to input mode. Input mode, however, only gives the user two options: typing in text, which is simply inserted into the file at the current line pointer, or pressing dual carriage returns, which returns the user to edit mode. (A blank line is entered by typing at least one space.) Even if the text that the user types in input mode is a command, it is not executed. To get into input mode, the user types the command input while in edit mode (if the file is new, the user is automatically placed in input mode on invocation of the editor). Often, a user might type a sequence like

```
locate/bull
next 3
type
```

only to discover, after some pondering, that the system is in input mode and that these commands were not executed but were actually inserted into the file as text. The "fix" is to get into edit mode, move the current pointer up n lines to the first erroneous line, delete n lines, move up 1 to reposition the pointer to the location from which the erroneous commands were first issued, and finally reissue the commands as commands.

Command	Page	Form	Arguments
Alter[1]	2-27	A	[range]
Copy[2]	2-40	C	position[=file-spec],range[,increment1[,increment2]] position=file-spec/C
Delete	2-43	D	[range]
End	2-44	E	[B] [Q] [S] [T] [file-spec]
Overwrite input file		EB	
No output file		EQ	
Strip line numbers		ES	
No numbers, no pages		ET	
Find[3]	2-46	F	[[string](ESC)[range]] [,A] [,N] [,E] [,n] [,-]
Help	2-51	H	[:n]
Input[4]	2-52	I	[position] [,increment] [position] [,increment] [position] [;!n]
Join	2-55	J	[position]
Kill page mark	2-56	K	/page
List:LP or file	2-57	L	[range[,S] [,P[:file-spec]]] [range] [,[S] [,F:file-spec]]
Mark	2-58	M	[position]
reNumber	2-59	N	[increment] [,[range] [,start]]
Print terminal	2-61	P	[range] [,S]
Replace[4]	2-63	R	[range] [,increment] [range] [,increment] [range] [,!n]
Substitute[5]	2-65	S	[[oldstring(ESC)newstring](ESC)[range] [,D] [,N] [,E]]
Transfer	2-69	T	position,range[,increment1[,increment2]]
Save World	2-73	W	[B] [file-spec]
eXtend[6]	2-74	X	[range] [,N]
Move Position	2-76		position
Give Parameter	2-77	=	parameter
Set Parameter	2-78	/	parameter[n]
Command File	2-79	@	file-spec
Print next line		(RET)	
Print previous line		(ESC)	

[1] Enter Alter mode
[2] Enter Copy-file mode (if /C).
[3] Enter Alter mode (if ,A).
[4] Enter Input mode.
[5] Enter Decide mode (if ,D).
[6] Enter Alter/insert mode.

Figure 1. SOS commands (From DIGI78. Copyright 1978 Digital Equipment Corporation. All rights reserved. Reprinted with permission.)

1.1.2 SOS

SOS [DIGI78], like the CMS editor, is a line editor designed for editing on "glass teletypes"—display terminals underutilized as hard-copy terminal emulators—on a time-sharing system, specifically a wide range of Digital Equipment Corporation computers. The input language is textual and is very similar to the CMS editor. The commands, as shown in Figure 1, are typed in prefix notation (verb/noun). The major unit of manipulation is the line.

Unlike the CMS editor, SOS attaches **fixed**, visible line numbers to each line in a file being edited. Typically a file is stored with these numbers, but special commands allow it to be stored without numbers and enable the numbers to be regenerated at the beginning of the next editing session. The editing buffer defaults to one line, although for most SOS commands a user can specify a line number or range of line numbers to expand this editing buffer. The default viewing buffer is a line; the window is

simply a mapping of this line to the output device.

For selection and organization purposes, SOS goes one step farther and allows the user to create logical pages within a file, using the page mark command. This essentially divides the file into subfiles that are independently sequence numbered. SOS maintains a current position pointer made up of the current page number and the current line number.

SOS is a highly modal editor, with the following seven different modes of operation (see Figure 2):

- **Input mode**, in which SOS accepts the text being typed and inserts it into the file;
- **Read-only mode**, in which a user can travel through a file but not modify it;
- **Edit mode**, in which the user spends much of the editing session performing editing, traveling, and viewing operations;
- **Copy-file mode**, in which the user can copy part or all of a file into another one;
- **Alter mode**, in which a user can perform character-by-character intraline editing without pressing carriage return to execute the command; it is a textual approximation to display editing without cursor keys;
- **Alter/insert mode**, in which the user can insert characters such as control characters that have special meaning to the editor;
- **Decide mode**, in which the user can make case-by-case decisions for substitute commands. In fact, decide mode has two submodes, **decide alter** and **decide alter/insert**. These two modes differ from **alter mode** and **alter/insert mode** primarily in that they, upon returning from the submodes, leave the user in **decide mode** rather than **edit mode**.

Alter mode is the most unusual of the modes. It simulates the intraline editing that is easily provided on display editors, and provides access to elements other than lines. The command syntax is postfix (noun/verb) and infix (noun/verb/noun), not prefix. Commands allow the user to skip forward and backward by characters and words, delete characters and words, capitalize and uncapitalize characters, delete all characters until the occurrence of a particular character, and so on. Unfortunately, the user must explicitly enter **alter mode** to take advantage of these facilities.

As Figure 2 shows, the transitions from mode to mode are almost mazelike; the user can easily become trapped in a remote area of the system. For instance, in **decide alter/insert mode** ESC brings one to **decide alter mode**, typing carriage return brings one to **decide mode**, typing CTRL-C brings one to **edit mode**, and typing CTRL-Y brings one to **DCL**, the operating system command interpreter. In **decide alter mode**, these command bindings change. While CTRL-C and CTRL-Y remain the same, now carriage return and linefeed bring the user to **decide mode**, and both I and R bring the user to **decide alter/insert mode**. In decide mode CTRL-C and CTRL-Y still perform the same, but E, Q, and G also bring the user to **edit mode**. This time A, as opposed to ESC, will bring the user to **decide alter mode**. The remaining transitions, as shown in Figure 2, are no less inconsistent and confusing.

Not only are the mode transitions difficult, but the actual command mnemonics for similar commands differ substantially from mode to mode. For example, in **edit mode**, the f (find) command allows the user to search for and move the editing buffer to the first line that contains a specified pattern; the s (substitute) command allows the user to replace an occurrence of an old pattern with a newly specified pattern. In **alter mode** the s now stands for skip and allows the user to find the next occurrence of a specified character; c (change) allows the user to change the next *n* characters in the line; f no longer exists.

SOS has some interesting concepts: powerful scope specification as a suffix to commands; a regular-expression pattern-searching facility, as shown in Figure 3; a query-replace user dialogue set up by **decide mode**; user-selectable toggles to indicate the level of experience of the user and to control the verbosity of prompts; and more. Yet the sheer complexity of the user interface often makes the system undesirable for even the most dedicated of users. We feel that SOS is a classic example of a

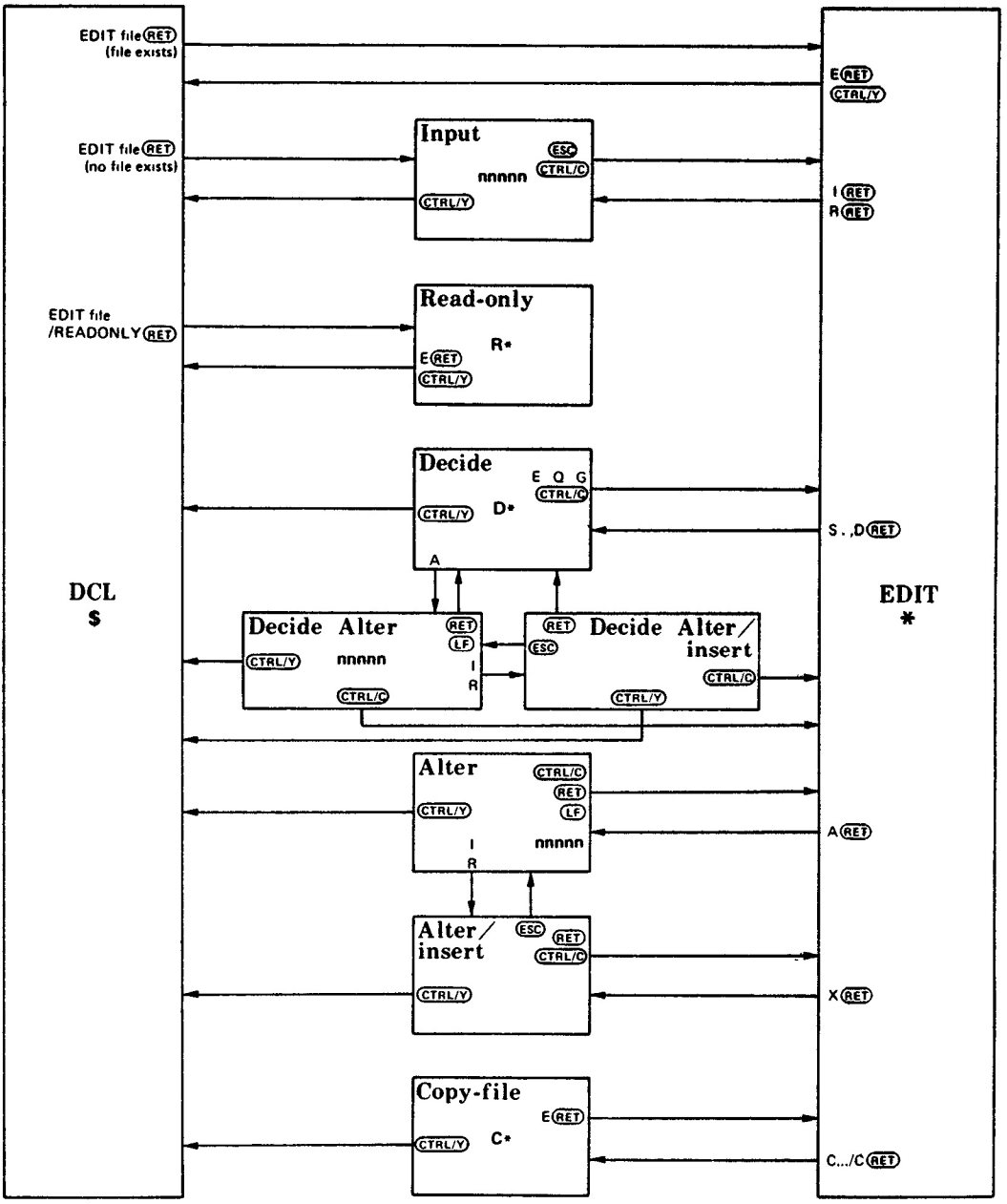


Figure 2. SOS modes. The paths among the various SOS modes and submodes of operation are marked by arrows. Prompts are shown in boldface type. (From DIGI78 Copyright 1978 Digital Equipment Corporation. All rights reserved. Reprinted with permission.)

powerful nucleus crippled by a poor user interface.

1.1.3 UNIX ed

The UNIX text editor, *ed* [KERN78a, KERN78b], is a variable-length line editor similar to both the CMS editor and SOS. *Ed*'s commands, like those in SOS, are only one or two characters long. It has a single-line viewing buffer but, like SOS, *ed* allows the user to expand the editing buffer for a command by specifying a range of line num-

Con-struct	Internal represen-tation	Meaning
Find-string constructs:		
?/	CTRL/T	Match any character
?:	|	Match any separator
?<	CTRL	Match a space or tab
?%x	CTRL/E	Match any character except x
?)x	CTRL/N	Match 0 or more of the character x
?1x	CTRL/V	Match 1 or more of the character x
?9	CTRL/X	Match any alphanumeric character
?!	CTRL/A	Match any letter (A-Z, a-z)
?&	CTRL/F	Match any uppercase letter (A-Z)
?2	CTRL/W	Match any lowercase letter (a-z)
?+	CTRL/P	Match any decimal digit (0-9)
?>	CTRL/]	Match beginning or end of line
?7c	CTRL/^	Match internal representation of c
Substitute-string constructs:		
?"	CTRL/B	Substitute next string matched
?*n?*	CTRL/O	Substitute nth string matched

Figure 3. SOS regular expression metanotation. (From DIGI78 Copyright 1978 Digital Equipment Corporation. All rights reserved. Reprinted with permission.)

bers in the form starting, ending as an optional prefix to each command. Thus, to perform the above change from add to subtract on the first 50 lines of a file, we use the *ed* substitute command:

```
1,50s/add/subtract/
```

The special *metacharacter* "$" indicates the last line of the file. Thus prepending a "1, $" to a command causes the buffer for that command to be the entire file. To move a number of lines, we simply say

```
1,10m/insert after this/
```

This will move lines 1 through 10 to follow the first line in the document that contains the string "insert after this". Lines in *ed* are of variable length so that truncation problems are solved.

A powerful feature of *ed* is its facility for user-specified regular expressions in patterns defining the scopes of operations (as opposed to other editors, which use regular expressions simply for search commands). (This feature has been available since NLS, but has become more common in other editors since its implementation in *ed* and SOS.) The user is supplied with the metacharacters

```
* $ ^ . [ ] \
```

with which to form regular expressions specifying the content of the pattern. The "*" is the Kleene star. Thus a character "n" followed by a "*" tells the editor to match the first character string containing a zero or more occurrences of "n". The "$" metacharacter in this use matches the end of the line, while the "^" caret companion matches the beginning of the line. A "." matches any character. The "\" escape character allows one of the metacharacters to be used as an actual character. Finally, the "[ ]" pair allows the user to specify a range of characters to be matched: [a-j] would match the first string (a single character) containing one of the letters lowercase "a" through lowercase "j"; [nkm] would match the first string with either an n or a k or an m. If the user wanted to find the first line beginning with a capital letter followed by a vowel in

the text of the Ogden Nash poem

```
I think that I shall never see
A billboard lovely as a tree
Perhaps unless the billboards fall,
I'll never see a tree at all.
```

the user would specify the search (using / as the find command)

```
/^[A-Z][aeiou]/
```

The ^ requires the pattern to be matched to start at the beginning of the line; the [A–Z] requires the first character of the pattern to be matched to be a capital letter; the [aeiou] requires the next character of the pattern to be matched to be a lowercase vowel. Upon executing this command from the top of the file, *ed* would find (and move the current pointer to)

```
Perhaps unless the billboards fall,
```

One interesting feature in *ed* is the ability to reference the scope of an operation indirectly in another operand of that operation. For instance, to parenthesize the entire line above, one would type

```
s/.*/(&)/
```

The ".*" is metanotation that means "match all characters on the current line." The "&" is metanotation that is shorthand for "all that were matched."

As in the CMS editor, lines in *ed* have varying internal numbers. Thus traveling is done as in the CMS editor, with both absolute and relative specifications and with context pattern specification as well. A time-saving feature is the use of a simple carriage return in edit mode (with nothing else on that line) as an implicit next 1 command. The user is given an explicit symbol called the *dot* to reference the current line pointer that can be used in arithmetic expressions to change the scope of an operation. For example,

```
.-10,.+7p
```

tells *ed* to print the 10 lines before the current position, the line at the current position, and the 7 lines after the current position. *ed* also allows the user to mark a specific line with a single lowercase character for later reference. The user simply types the save position command (abbreviated k) followed by a single character label, as in

```
kx
```

and the current line is now referenced with "x". To travel to that line, the user simply types "' ", the goto saved-position command, followed by the label

```
'x
```

and immediately is returned to that saved position. Like the CMS editor, *ed* has two main modes, edit mode and append mode, and the associated problems of two modes. In fact, these problems are compounded by the fact that *ed* is, as characterized by Norman, "shy":

> Ed's major property is his shyness; he doesn't like to talk. You invoke Ed by saying, reasonably enough, "ed." The result is silence: no response, no prompt, no message, just silence. Novices are never sure what that silence means. Ed would be a bit more likeable if he answered "thank you, here I am," or at least produced a prompt character, but in UNIX, silence is golden. No response means that everything is okay; if something had gone wrong, it would have told you. [NORM81, p. 144. Reprinted with permission of *Datamation*® magazine, ©copyright by TECHNICAL PUBLISHING COMPANY, a DUN & BRADSTREET COMPANY (1981), all rights reserved.]

In the edit/append mode dichotomy, this silence causes major confusion. To add text to the file, the user issues the "a" command to append, followed by a carriage return. Unlike the CMS editor, *ed* gives no indication that it is now in append mode; it just waits for the user to input text, like the CMS editor. To return to edit mode, the user types a line with only a "." on it and follows it with a carriage return. As Norman points out, this is not an oversight, but in fact is acknowledged, rather flippantly, in the documentation:

> Even experienced users forget that terminating "." sometimes. If ed seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later. [KERN78a, p. 2]

One of the designers of UNIX system software defends the terseness of UNIX commands by citing their contribution to an important capability of UNIX: the ability to easily use the output of one program as the input to another [LESK81]. But silencing a user-oriented interactive program

so that its output may be used by another program seems to us a large price to pay. In fact, UNIX easily allows the user to select just which output should be passed on to another program as standard input; careful programming can ensure that user prompts and status information can be interspersed with standard output without interfering with it.

While *ed* is a powerful line editor, it is questionable whether the interface, which requires the user to memorize small, nonmnemonic, and often obscure command names and, more critically, to "guess" the status of the system, is proper for a general-purpose audience. In fact, this editor was developed not for a large community, but for a group of a half-dozen computer science researchers familiar with the notions of regular expressions and file organization who were designing the operating system and file system in which the editor would run; they wanted maximum keystroke efficiency and minimum distraction. While the *ed* line editor has illuminated several important concepts in editing, it nevertheless represents a decreasingly popular breed of editors.

1.2 Stream Editors

Stream editors act upon a document as a single, continuous chain of characters, as if the entire document were a single, indefinitely long character string, rather than act upon fixed-length or variable-length lines. By doing so, they avoid line editor problems such as truncation and inability to perform interline searching or editing. TECO, described below, is the most popular editor of this category.

1.2.1 TECO

TECO, the Text Editor and COrrector (ca. 1970), is an interpreter for a string processing language. TECO can be used interactively as a stream-oriented editor; its basic commands can also be used as building blocks to provide quite elaborate editing operations. Many variations exist (DEC TECO and TENEX TECO are two), with varying capabilities and syntax. The conceptual model considers a document to be a sequence of characters, possibly broken into variable-length virtual pages by form-feed characters, and into virtual lines by line-end characters. Pages may be combined in an in-core editing buffer considered to be simply a varying-length string whose length may grow up to the in-core memory available.

The interface is based on typed input, typically consisting of single-character command syntax of the form

[argument][single character command]

Commands can be combined to form sequences. Regardless of whether the user specifies a single or combined command, TECO does not interpret the command string until the user presses the ESC key. In the ensuing examples, the terminating ESC is implied. The editing buffer is the amount of the file in memory. The viewing buffer on the document defaults to the null viewing buffer. The document is displayed only upon explicit command; the user can specify a viewing buffer of any size, as explained below.

TECO maintains the current position as a value called point (symbolized by "."), which simply contains the number of characters in the buffer to the left of it. This pointer can be positioned absolutely (by a numeric value), relatively (by a positive or negative character or line displacement), or by pattern searches. For example, in TENEX TECO [BBN73], 0J or BJ jumps the pointer to the top of the buffer, ZJ jumps the pointer to the end of the buffer, 43J jumps to the 43rd character of the buffer, .−9 or −9C moves the pointer backward nine characters, and 17;BJ jumps to the top of page 17. The symbols Z, B, and . are not simply command modifiers but are registers that contain the point for the end of the buffer, the point for the beginning of the buffer, and the current point, respectively; thus the above commands using these registers resolve to an absolute character address.

Although TECO is character oriented, special commands allow the user to edit a document in terms of a line model. Again, using appropriate register values, L moves the pointer to the beginning of the next line, −L moves the pointer to the beginning of the previous line, 0L moves the pointer

to the beginning of the current line. Similarly, :L moves to the end of the current line, while −:L moves to the end of the previous line. Line-oriented printing commands are provided as well; 7T prints the characters from the pointer until the beginning of the seventh line after the pointer, T prints the segment of the current line after the pointer. We stress that lines are an abstraction provided to the user; the text is stored not as lines, but simply as sequences of characters that are interpreted as lines by a filter that understands special line-end delimiters. A more complicated filter, for instance, might be able to extract programming language constructs from the stream.

Fundamental commands such as insert, delete and context search are supplied. INow is the time(CTRL-D) inserts the string "Now is the time" before the current pointer. 5D deletes the five characters after the pointer. Sgood(CTRL-D) finds the first match for "good" after the current pointer and moves the pointer there; similarly, Rgood(CTRL-D)bad(CTRL-D) replaces the first occurrence of "good" with "bad."

Importantly, TECO also supports commands for conditional execution to aid in creating more complex commands. *Q-registers* are available for holding any numeric or string value. Simple uses include performing arithmetic and moving or copying strings. To move a string of text, for example, the string is first saved in a Q-register and then deleted from the buffer (in some versions, the deletion is automatic). Next, the character pointer is moved to a new location and the contents of the Q-register are copied into the buffer at this new point.

If a Q-register contains text, the text may be interpreted as a command string. Thus, TECO can be used as a programming language to build editing commands. Higher level commands are created by joining together many lower level operations. Consider the pseudocode for a global change operation with query and replace prompting:

```
WHILE (pattern is found in source)
   IF user response = "Y" THEN
      substitute newstring for pattern
END
```

With this pseudocode in mind, to query and replace "good" with "bad," one could write the TECO code

```
J⟨Sgood(CTRL-D) ;V↑T-↑↑Y"E-4C
Rgood(CTRL-D)bad(CTRL-D)'⟩
```

J puts the pointer at the beginning of the buffer. The ⟨ ⟩ pair are loop delimiters, indicating that the commands inside the loop should be executed repeatedly. Sgood(CRTL-D) is the search command that we have seen previously. Upon failure of the search, ; V skips to the end of the loop construct. The ↑T is a "variable" that is assigned the value of a character typed by the user, while the ↑↑Y is the value of a capital Y. The subtraction expression, ↑T-↑↑Y, equals zero only when a Y is typed in. Thus, if the preceding expression is equal to zero, then the commands following the E are run; otherwise everything until a delimiting ' is skipped. The -4C moves the pointer to just before the beginning of good. Finally, the Rgood(CTRL-D)bad(CTRL-D) performs the appropriate replacement. The loop then repeats until failure.

The raw power of TECO is evident from the above example. The abstraction of text (a continuous stream of text with a pointer) is simple, especially for the programmer, as it parallels the abstraction of computer memory with associated program counter. Continuing this analogy, TECO is to a text stream what assembly language is to sequential computer memory. The TECO language provides a powerful base for a trained systems programmer or for a compiler's code generator; however, it does not provide a reasonable high-level interface for the average user, just as assembly language does not provide a reasonable interface for the casual (and even proficient) programmer. The syntax is cryptic. While all commands operate at the point, user misconceptions of the exact point location often result in off-by-one errors. TECO has been used effectively as an implementation language in several editors, most notably in EMACS, described below. However, we believe that it is not a proper tool for either knowledge workers or competent programmers because of its low-level orientation.

1.3 Display Editors

This category includes several editors based on work done by Deutsch [DEUT67]

and on the work of Irons and Djorup [IRON72], as well as several editors with an Irons-like model. The simple Irons outline for a CRT editing system has been the backbone of many editors: NED [BILO77, KELL77], *bb* [REIS81], PEN [BARA81], Z [WOOD81], and sds [FRAS81]. We present a general overview of the standard functions available in this kind of editor and then describe in more detail the unique features of several specific instances.[2]

In the Irons conceptual model, text is conceived of as a quarter-plane extending indefinitely in width and length, with the topmost, leftmost character the origin of the file. The user travels through this plane by using cursor keys and changes characters by overtyping. At any time, the user sees an accurate portrayal of the portion of the file displayed. Text is input on the screen at the position of the cursor. The environment is "modeless"; since all typing on the screen is considered text, commands must be entered either through function keys, control characters, and escape sequences, or by moving the cursor to and typing in a special command line at the bottom of the screen.

The command syntax is typically single-operand postfix. Basic traveling and editing primitives are provided, such as +/− pages, +/− lines, +/− words, insert character, delete word, and back word. Some of these may be preceded by an optional modifier. Thus, + page scrolls forward to the next page, while 3 + page scrolls three pages. Additionally, the editing and viewing buffers can be moved left and right and multiple windows support easy interfile editing. These editors make use of pick and delete buffers; hence deleted text is not discarded but is put in a buffer for possible subsequent use for moving or copying text. Functions such as delete, pick, and put may be combined with element modifiers such as character, word, line, and paragraph to allow more familiar specification of deletes, copies, and moves. A marking facility allows the user to select with the cursor two arbitrary points in the text to define a scope not easily specified with the element modifiers.

For display, most of the Irons derivatives use special algorithms to minimize the amount of screen updating necessary.

1.3.1 Brown's bb

Brown's *bb* [REIS81] is a typical example of the Irons model editor. Running under the UNIX operating system on a VAX 11/780, it makes use of a wide range of function keys for interaction.

One of *bb*'s extensions of the model is the maintenance of an up-to-date temporary file on disk along with a linked list of changes that have been made to the old file. This change history serves as the backbone of the undo command, which is capable of reverting changes back to the beginning of the editing session.

For travel, as well as providing the standard +/− keys, *bb* allows the user to save positions in named buffers and to jump to these positions with a goto command.

bb provides user manipulation of the instructions with the do facility. Rather than providing a macro language, do provides a mechanism for capturing and naming a group of keystrokes. In general, a *programming-by-example* facility is an extremely elegant, powerful tool for both the novice and the experienced user. The user does not have to think in terms of a macro language syntax (with associated variables, flow-of-control constructs, and textual verbosity), but defines the new operations in terms of the same syntax that is used for editing. Complex operations that are hard to specify in a procedural macro are almost trivial in terms of keystroke macros, where the user simply executes the commands while the system captures them. For example, to find all instances of a *troff/me* italic formatting command—a separate line of the form

```
.i "this will be italicized"
```

—and change them to the TEX form

```
{\sl this will be italicized}
```

one could use the following keystroke macro (all capitalized words are commands implemented as function keys or control

[2] In this general overview, the syntax used does not reflect that of any given system, nor does an example of a general operation imply that each system contains that operation

sequences):

```
TOF                          [goes to top of file]
DOBEG                [begins capturing keystrokes]
ENTER ^.I +REG-EXPR-SEARCH
                    [search forward for an
  occurrence of ".I" which starts on a new line]
{\sI                 [type over existing ".I ""]
INSERT-SPACE         [inserts needed space]
+EOL [goes to end of line, 1 character past the
                                        quote]
BACKSPACE            [put cursor at end quote]
}                  [types right bracket over quote]
DOEND              [finishes capturing keystrokes]
```

Now every time the user presses the DO key, *bb* will perform all the keystrokes entered between the DOBEG and DOEND keys. *bb* does not support parameterized keystroke macros or macros that prompt for particular input and subsequently continue executing; hence one could not design a general-purpose keystroke macro similar to the special-purpose one above.

bb examines the file extension (file type) of the current file and loads an internal table with target-dependent information. This allows *bb* to perform automatic indenting for various programming languages and to recognize structural entities such as paragraphs in documents. Like many of the editors in this category, *bb* supports multiple viewing buffers and windows, although it only maintains a single editing buffer.

bb allows users to bind their own personal keyboards to the standard commands by modifying a control file. *bb* also supports an invocation time profile, allowing personalized defaults on startup. This is coupled with a state-save facility that maintains necessary parameters from session to session. A help facility allows easy access to a complete online manual. Screen manipulation is performed by looking up terminal capabilities in the UNIX *termcap* database [JOY81] to determine output device characteristics, and by using specialized screen-optimization algorithms.

1.3.2 Yale's Z Editor

Yale's Z editor [WOOD81] extends the general Irons functionality by providing facilities that aid in program creation while maintaining the general-purpose functionality of the editor.

Editor commands are entered using control characters coupled with the cursor keys. Function keys are not used; the developers dislike the fact that the user's hand must be moved from the typewriter keyboard to use them. Software allows overloading of the standard ASCII character set by using certain keys as shift keys. The interaction language also supports the overloading of each editor command. Here, as in most of the Irons derivatives, one command may be made to do slightly different things by prefacing it with optional arguments. For example,

arg *string* fSearch

searches forward for the next instance of the pattern *string* and moves the cursor there if successful. Each command may be prefixed with the special command meta, slightly altering the function of the command to which it is attached. For instance, meta fSearch causes case-insensitive searching.

For travel, Z remembers the last seven buffer positions, allowing the user to review previous contexts while the current one retains the status quo. Like *bb*, Z allows the user to put a "bookmark" on a certain spot in the file for later return.

The unique features of Z are its solutions to the program-editing task. Rather than using the structure-oriented approach, in which the editor has specific knowledge of the syntax (and possibly the semantics) of a target-programming language, the Z editor represents programs as text, offering visual cues and a tight interface with existing compilers and debuggers to take the place of the innate knowledge of syntax-directed editors.

The designers of Z contend that "existing structure-oriented program editors have several disadvantages, such as increased complexity in the implementation, a restrictive user interface, and poor support for editing" [WOOD81, p. 4]. Their solution is to represent the program as a text while equipping the editor with knowledge of program elements such as quoted strings, end-of-line delimiters, and matching tokens (such as begin/end) that signal indentation, as well as with indentation rules.

This representation allows Z to perform many functions normally associated with structure editors. Prettyprinting for block-

structured languages is done by examining the last token on a line when the newline key is pressed. If that token is in a table that lists it as a token requiring subsequent indentation or exdentation relative to the previous line, the next line will be appropriately indented or exdented. This algorithm gives the desired result most of the time; a manual mode is offered to correct any mistakes made by the automatic mode. The matching token table also drives commands that close off the most recently begun matching unit, that find the end of the nearest unit if already closed, and that skip over the matching expressions as single units. This is particularly useful in LISP programming, where levels of parenthesization are hard to manipulate. Again, this facility does not require the editor to have syntactic knowledge of the target language, but simply to maintain a table of matching tokens.

Syntax-directed structure editors allow the user to manipulate syntactic units as single entities, as well as to view levels of syntactic detail. Z provides analogous features based on indentation level, which the designers claim work because "all the important information about the block structure of a program is contained in the indentation, provided the programmer is consistent" [WOOD81, p. 5].

The designers of Z chose to do neither the syntactic checking nor the incremental compilation often associated with syntax-directed editors, as the philosophy is **not** to integrate the compiler directly into the editor. Feeling that "the programmer is the person best able to decide when his program is in a state ready for compilation" and that "existing compilers are perfectly able to locate errors" [WOOD81], the designers of Z attempted to enhance communications between editor and compiler. The user can execute the compile command from the editor; this communicates with an asynchronous process that formats the compiler request per the target language, puts the request on the processor queue, appends error messages in a special file, and returns a completion message to the editor when done. The user can continue editing while this is being done.

Z also provides a link to Multiple User Forks, a program that maintains multiple user contexts in parallel. This allows the user to exit from Z into any of the other forks (perhaps to read documentation or check on the state of some running program) and to return to Z without loss of state.

1.3.3 EMACS

EMACS is an M.I.T. display editor designed to be "extensible, customizable, and self-documenting" [STAL80, STAL81]. Several versions and dialects exist, most notably the Stallman version for the Tops-20 operating system (from which our examples are derived), the Honeywell Multics version [GREE80], and the UNIX version by James Gosling of Carnegie-Mellon. Some versions of this full-screen editor for time-sharing systems are written in TECO; others are written in the high-level language LISP. To extend or customize the functionality, users write routines in the same language as that in which the standard editor functions are written, rather than using an editor macro language. Richard Stallman, the designer of the first EMACS, feels that this capability allows the user to transcend any limitations imposed by the editor's implementors. The basis of the successful EMACS strategy is that defining the extensions or changes in the source language "is the only method of extension which is practical to use" [STAL81, p. 147]; it is unwise to maintain a "real" implementation language for the implementors and a "toy" one for the users. The user is able to bind many extensions or changes in a library, which can be loaded at invocation time. In fact, many of the core facilities that exist today were originally user-written extensions and were later adopted into the production system, encouraging arbitrary growth rather than design. EMACS does offer a keystroke macro facility with prompts, so that nonprogramming users do have an alternative to a programming language at their disposal.

In EMACS, every typed character is considered a command. The keys for printing characters are bound, by default, to a command called self insert that causes that character to be inserted into the text at the cursor location. Generally, nonprinting characters (control and escape sequences)

invoke commands to modify the document. The editing language accepts single-character commands and finds the current binding between command key and function in a table. EMACS and other display editors offer a quote facility that allows characters typically used as commands to be inserted as characters into a document. A few of its many interesting features include a query replace facility, transposition functions for lines and words, and an automatic balanced parenthesis viewer.

The windowing facilities are extensive as well. The system supports multiple open files and hence editing buffers[3] with associated viewing buffers and windows. The CTRL-X 2 command divides the screen into two windows containing the viewing buffers designated by the user. The cursor is in one window at a time; CTRL-X O switches between windows. Special "narrowing" commands serve to change the size of the viewing buffer and editing buffer while leaving the window intact. The user marks one end of a region, moves the cursor to the other end, and issues the command CTRL-X N. This range now defines the maximum range of the editing buffer. If it is larger than a window's worth of text, the viewing buffer is set to a full window's size; otherwise, it is set to the size of the editing buffer.

The fact that all keys (including alphanumerics) are bound to actions is very important, as the self insert action can be extended to effect more complex results. For example, one can extend the definition of the space character to insert itself and to check to see if an automatic word wrap is necessary. A more detailed use of the key redefinition facility is the EMACS abbreviation package. Here, all the punctuation characters are redefined to look at the previous word, to check for its existence in an abbreviation table, and if it exists, to substitute the expanded word for the abbreviation as the user types.

EMACS offers **major modes**, editing environments tailored for editing a particular kind of file. For example, **text mode** treats the hyphen as a word separator; **LISP mode** distinguishes between lists and s-expressions. The major mode can automatically define different key bindings for a particular application. For example, in many programming language major modes, the tab key is redefined to provide automatic indentation.

EMACS can keep a record of all keystrokes typed in a session in a *journal file*. If a system crash destroys a current editing session, the user can instruct EMACS to bring up an old version of the file and replay the keystrokes from the journal file. The user watches the changes being made, and can stop the process at any time. (This allows a primitive undo facility: the user can replay up to a desired point and then discard the rest of the changes that are no longer wanted.)

Another interesting facility for program editing is the TAGS package. The separate program TAGS builds a TAGS table containing the file name and position in that file in which each application program function is defined. This table is loaded into EMACS; specifying the command Meta, *function name* causes EMACS to select the appropriate file and go to the proper function definition within that file. Other special libraries include DIRED, a subsystem for editing a file system directory using the full-screen display capabilities, and BABYL, a complete message-handling subsystem. INFO reads tree-structured documentation files, performing the necessary operations to travel from one node to the next.

EMACS has a very large and faithful following in the academic research community. While the basic editor is not vastly different in functionality from the Irons model editors, the customized, application-specific packages have "sold" the system. To obtain a distribution of EMACS, one must agree to redistribute all extensions that one develops. By now, these extensions are quite numerous and powerful. Thus it is not the raw editor, but the editor and its extensions that far exceed the capabilities of most other editors. While the programming language certainly cannot be used competently by the average user, the availability of extensibility features for programmers has manifested itself in many powerful facilities.

[3] Note that our use of the term "editing buffer" here, while consistent with our previous usage, differs slightly from Stallman's terminology in describing EMACS [STAL80, STAL81].

Extensibility, however, has negative points. Although the major new packages are distributed to all EMACS installations, many customizations are personal ones. This leads to situations in which two people using EMACS have different syntax and functionality. One set of keyboard bindings might be different from the next (e.g., one CTRL-D moves down one line, while another deletes a line) or, alternatively, an identical keyboard arrangement and apparently identical functionality may have fine distinctions that confuse a user from a different microcosm trying to use someone else's EMACS (e.g., one GOTO END-OF-WORD command might go to the last character in a word, preceding punctuation, while another may go the first white space following the word). Thus with extensibility in any editor comes the price of widespread divergence over various installations and even over the same installation. The trade-off between a large number of divergent but customized dialects and a single, standard language is unclear.

1.3.4 IBM's XEDIT

XEDIT [IBM80] is IBM's screen editor for their VM/CMS time-sharing system. Unlike the Irons model editors just described, XEDIT uses local terminal intelligence to perform screen-editing operations. The high-level conceptual model is the unbounded quarter-plane of text in which the user views a rectangular region, yet XEDIT still relies on the sequential card model in some of its operations.

The display editing functions of XEDIT only work on IBM 3270 or 3270-compatible terminals. These terminals have a local screen buffer memory and a special keyboard with keys that support editing on the local buffer, such as add char, delete char, and delete to end of line.

The user has several methods of command specification. The first is changing the screen image by driving a cursor and using the local editing capabilities. The user is able to edit both the displayed text and a status line that displays file name, record length, and several other options. The user changes the text and the status line by simply inserting, deleting, or changing the options field of each line. Pressing the ENTER key causes the contents of the editing buffer to be sent to the host computer, which determines the difference between the terminal's local editing buffer and the host's internal data structure, and updates its internal data accordingly. This synchronization between the screen buffer and the internal data structure is an important component of an editor using a terminal with local intelligence. Note that throughout this editing process, the host processor is not signaled of any changes, regardless of how long the user has been editing a screenful of text, until the user presses the ENTER key to transmit that screen.

Besides the display-editor-style commands, XEDIT accepts typed commands that are almost identical to those of the CMS line editor; in fact, on non-3270 series terminals, XEDIT operates essentially like the CMS line editor. These commands, typed in on a special command line, control those operations that cannot be done using the local editing buffer, including control functions, such as reorganizing viewing buffer–window mappings and ending a session, search commands, and some types of insert, move, and copy commands. Any of these commands can be bound to the ten function keys on the keyboard. XEDIT cannot support selection by marking with a cursor because the current position of the cursor cannot be read by the CPU. While it does allow textual specification of region commands, it also provides the user with a **prefix** field before each line on the screen (see Figure 4) to give additional functionality.

As we see in Figure 4a, the D on the line beginning "THE HIPPOPOTAMUS" marks this line for subsequent deletion. The 2a is an instruction to add two blank lines after this line. The DD is a grouping marker that delimits the beginning and end of a region of text to be deleted (the two DDs need not be on the same screen of text). The single A again stands for adding a blank line. When the user presses ENTER, the screen buffer is transmitted, and the host computer interprets the prefix fields (as well as any local editing), updates the internal data structure appropriately, and redisplays the updated text, as shown in Figure 4b.

```
ANIMALS  FACTS    A1  F 80  TRUNC=80 SIZE=14 LINE=9 COLUMN=1

===== * * * TOP OF FILE * * *
D==== THE HIPPOPOTAMUS IS DISTANTLY RELATED TO THE PIG.
===== ELEPHANT TUSKS CAN WEIGH MORE THAN 300 POUNDS.
===== LAND CRABS FOUND IN CUBA CAN RUN FASTER THAN A DEER.
===== ELECTRIC EELS CAN DISCHARGE BURSTS OF 625 VOLTS,
=2a== 40 TIMES A SECOND.
===== THE ANCIENT ROMANS AND GREEKS BELIEVED THAT BEDBUGS HAD MEDICINAL
===== PROPERTIES WHEN TAKEN IN A DRAFT OF WATER OR WINE.
==DD= STURGEON IS THE LARGEST FRESHWATER FISH AND CAN WEIGH 2250 POUNDS.
===== ANTS HAVE FIVE DIFFERENT NOSES.  EACH ONE IS DESIGNED TO
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
=DD== ACCOMPLISH A DIFFERENT TASK.
==A== ALL OSTRICHES ARE POLYGAMOUS.
===== SNAKES LAY EGGS WITH NONBRITTLE SHELLS.
===== THE PLATYPUS HAS A DUCK BILL, OTTER FUR, WEBBED FEET, LAYS
===== EGGS, AND EATS ITS OWN WEIGHT IN WORMS EVERY DAY.
===== * * * END OF FILE * * *



===>                                                    X E D I T  1 FILE
```

```
ANIMALS  FACTS    A1  F 80  TRUNC=80 SIZE=13 LINE=9 COLUMN=1

===== * * * TOP OF FILE * * *
===== ELEPHANT TUSKS CAN WEIGH MORE THAN 300 POUNDS.
===== LAND CRABS FOUND IN CUBA CAN RUN FASTER THAN A DEER.
===== ELECTRIC EELS CAN DISCHARGE BURSTS OF 625 VOLTS,
===== 40 TIMES A SECOND.
=====
=====
===== THE ANCIENT ROMANS AND GREEKS BELIEVED THAT BEDBUGS HAD MEDICINAL
===== PROPERTIES WHEN TAKEN IN A DRAFT OF WATER OR WINE.
===== ALL OSTRICHES ARE POLYGAMOUS.
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
=====
===== SNAKES LAY EGGS WITH NONBRITTLE SHELLS.
===== THE PLATYPUS HAS A DUCK BILL, OTTER FUR, WEBBED FEET, LAYS
===== EGGS, AND EATS ITS OWN WEIGHT IN WORMS EVERY DAY.
===== * * * END OF FILE * * *



===>                                                    X E D I T  1 FILE
```

Figure 4. Add and delete with XEDIT prefix commands. (From IBM80, Reprinted by permission of the IBM Virtual Machine/System Product: SYSTEM PRODUCT EDITOR'S GUIDE (SC24 5220-0). © 1980 by International Business Machines Corporation.)

Figure 5 shows a similar move command using the prefix fields. The mm is a grouping marker, much like the DD above, to delimit the beginning and end of a multiline region of text to be moved, and the f is a marker signaling the line after which the moved text should be inserted.

XEDIT's local editing style offers both advantages and disadvantages. The use of the local 3270-series editing capabilities implies that users need not worry about an overloaded host system most of the time; most of the intraline editing, and even some of the block moving, as above, can be done without intervention of the host CPU. The editor is dependent upon the host system only when a screenful of text must be transmitted or a textual command (like search) must be executed. On the other hand, the local buffer offers no safety; if the host system crashes while a user is screen editing, all modifications on the local buffer are lost. More specific to XEDIT, the inability to do region selection within lines (because marking without CPU intervention is impossible on the 3270) reduces the generality of the editor. Additionally, the several styles of commands (typed, cursor driven, prefix field) can confuse a novice user.

1.4 Graphics-Based Interactive Editor/ Formatters

1.4.1 Xerox PARC's Bravo

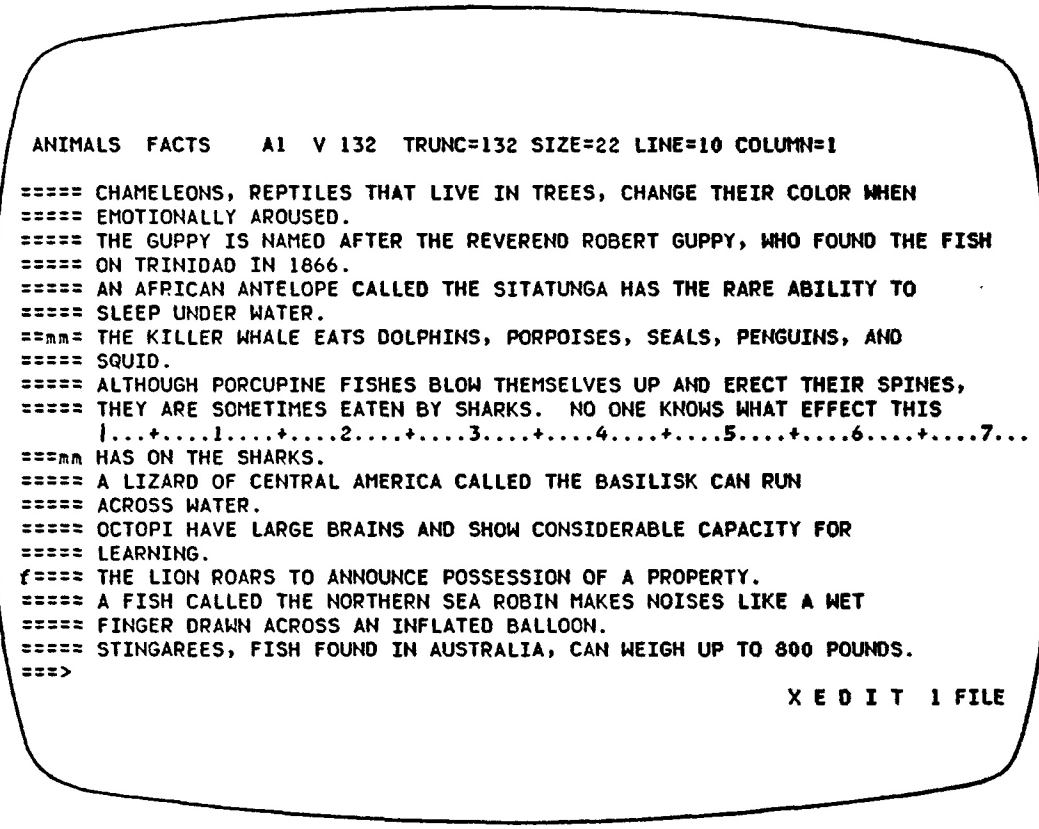
Xerox PARC's Bravo (ca. 1975) is one of the first of the interactive editor/formatters based on the display of high-resolution, proportionally spaced text. Bravo allows the creation and revision of a document containing soft-typeset text with justification performed instantly by the system. The conceptual model is of a continuous scroll of typeset text that can be paginated when desired.

Bravo runs on Xerox's Alto, a 16-bit minicomputer with a raster graphics "portrait" display (roughly $8\frac{1}{2} \times 11$ aspect ratio) of 606×808 pixel resolution. This high-resolution pixel-addressable display allows more complex visual cues (overlapping windows, typeset facsimile text, graphics) than does the alphanumeric CRT terminal. A mouse drives a cursor and offers three buttons (called left, middle, and right) that can be read independently of the cursor.

Bravo offers a mix of graphical and keyboard user interfaces. By moving the mouse, the user drags the cursor across the screen. The cursor addresses characters, special "menu" items, and other selectable elements on the screen. The interaction language is modal: the user can be in either *command mode*, in which text elements can be selected and commands initiated, or *typing mode*, in which keyboard text is entered into the document. Because of the modes, the user can specify commands with single alphanumeric characters; unlike many display editors, alphanumeric characters are not entered into the document unless the user is in typing mode. Commands not invoked with single alphanumeric keys are invoked with control characters.

As shown in Figure 6, the Bravo screen is divided into several areas. The *system window* contains information concerning what the user has just done and what can be done at present. The *document window* contains a viewing buffer's worth of the document text scroll. The *line bar* and *scroll bar* are graphical entities that help the user travel through the document.

To travel in Bravo, the mouse is used to move a double-headed arrow cursor along the scroll bar, a vertical strip at the left side of the document. Pressing the left button on the mouse while the arrow is pointing to a line in the document's window causes that line to become the top line in the window; pressing the right button causes the top line in the window to move to the line the cursor is at. For more extensive traveling, one is supplied with a graphical *thumbnail* that moves along the scroll bar, and a *bookmark* that indicates the approximate current position in the document on a graphically displayed linear continuum from "front cover" to "back cover." If the user is halfway through the document, for instance, the bookmark indicates a point halfway across the continuum. To travel to a part of the document preceding what is being viewed, the user simply places the thumbnail somewhere before the bookmark on the continuum and presses the middle button; the document will "fall open" at the corresponding position in the text. By plac-


ANIMALS FACTS A1 V 132 TRUNC=132 SIZE=22 LINE=10 COLUMN=1
===== CHAMELEONS, REPTILES THAT LIVE IN TREES, CHANGE THEIR COLOR WHEN
===== EMOTIONALLY AROUSED.
===== THE GUPPY IS NAMED AFTER THE REVEREND ROBERT GUPPY, WHO FOUND THE FISH
===== ON TRINIDAD IN 1866.
===== AN AFRICAN ANTELOPE CALLED THE SITATUNGA HAS THE RARE ABILITY TO
===== SLEEP UNDER WATER.
==mm= THE KILLER WHALE EATS DOLPHINS, PORPOISES, SEALS, PENGUINS, AND
===== SQUID.
===== ALTHOUGH PORCUPINE FISHES BLOW THEMSELVES UP AND ERECT THEIR SPINES,
===== THEY ARE SOMETIMES EATEN BY SHARKS. NO ONE KNOWS WHAT EFFECT THIS
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
===mm HAS ON THE SHARKS.
===== A LIZARD OF CENTRAL AMERICA CALLED THE BASILISK CAN RUN
===== ACROSS WATER.
===== OCTOPI HAVE LARGE BRAINS AND SHOW CONSIDERABLE CAPACITY FOR
===== LEARNING.
f==== THE LION ROARS TO ANNOUNCE POSSESSION OF A PROPERTY.
===== A FISH CALLED THE NORTHERN SEA ROBIN MAKES NOISES LIKE A WET
===== FINGER DRAWN ACROSS AN INFLATED BALLOON.
===== STINGAREES, FISH FOUND IN AUSTRALIA, CAN WEIGH UP TO 800 POUNDS.
===>
XEDIT 1 FILE

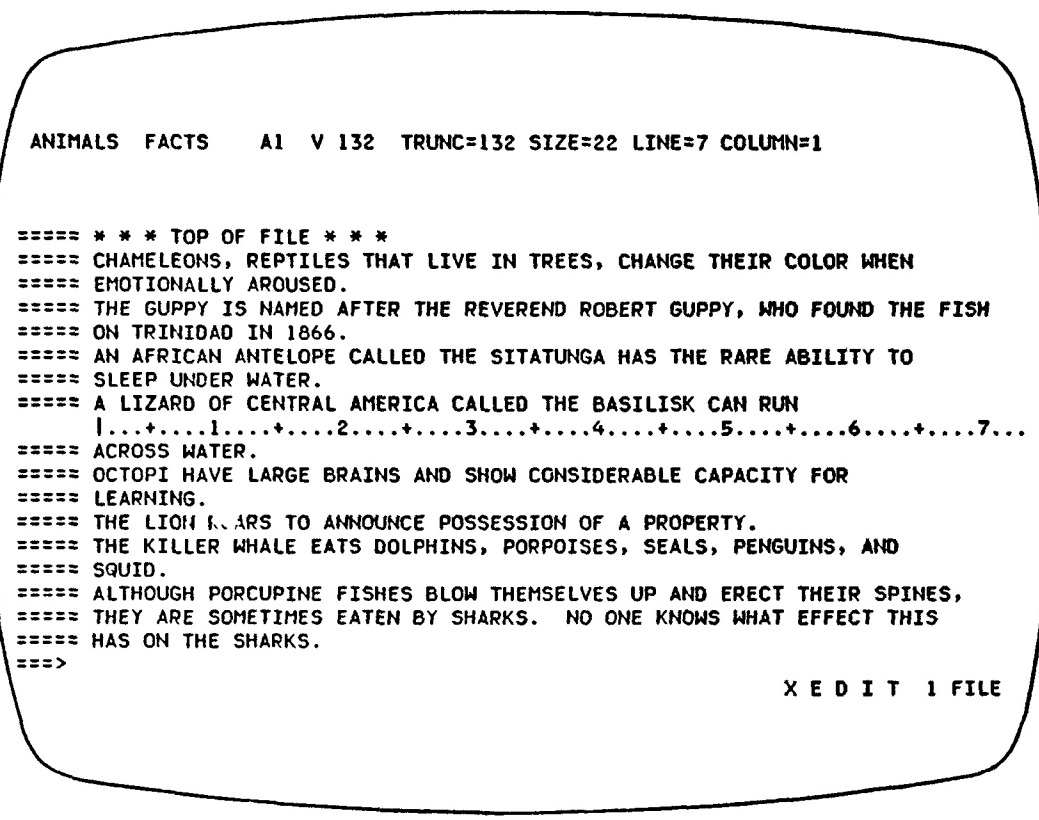

ANIMALS FACTS A1 V 132 TRUNC=132 SIZE=22 LINE=7 COLUMN=1
===== * * * TOP OF FILE * * *
===== CHAMELEONS, REPTILES THAT LIVE IN TREES, CHANGE THEIR COLOR WHEN
===== EMOTIONALLY AROUSED.
===== THE GUPPY IS NAMED AFTER THE REVEREND ROBERT GUPPY, WHO FOUND THE FISH
===== ON TRINIDAD IN 1866.
===== AN AFRICAN ANTELOPE CALLED THE SITATUNGA HAS THE RARE ABILITY TO
===== SLEEP UNDER WATER.
===== A LIZARD OF CENTRAL AMERICA CALLED THE BASILISK CAN RUN
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
===== ACROSS WATER.
===== OCTOPI HAVE LARGE BRAINS AND SHOW CONSIDERABLE CAPACITY FOR
===== LEARNING.
===== THE LION ROARS TO ANNOUNCE POSSESSION OF A PROPERTY.
===== THE KILLER WHALE EATS DOLPHINS, PORPOISES, SEALS, PENGUINS, AND
===== SQUID.
===== ALTHOUGH PORCUPINE FISHES BLOW THEMSELVES UP AND ERECT THEIR SPINES,
===== THEY ARE SOMETIMES EATEN BY SHARKS. NO ONE KNOWS WHAT EFFECT THIS
===== HAS ON THE SHARKS.
===>
XEDIT 1 FILE

Figure 5. Move with XEDIT prefix commands. (From IBM80. Reprinted by permission of the IBM Virtual Machine/System Product: SYSTEM PRODUCT EDITOR'S GUIDE (SC24 5220-0). © 1980 by International Business Machines Corporation.)

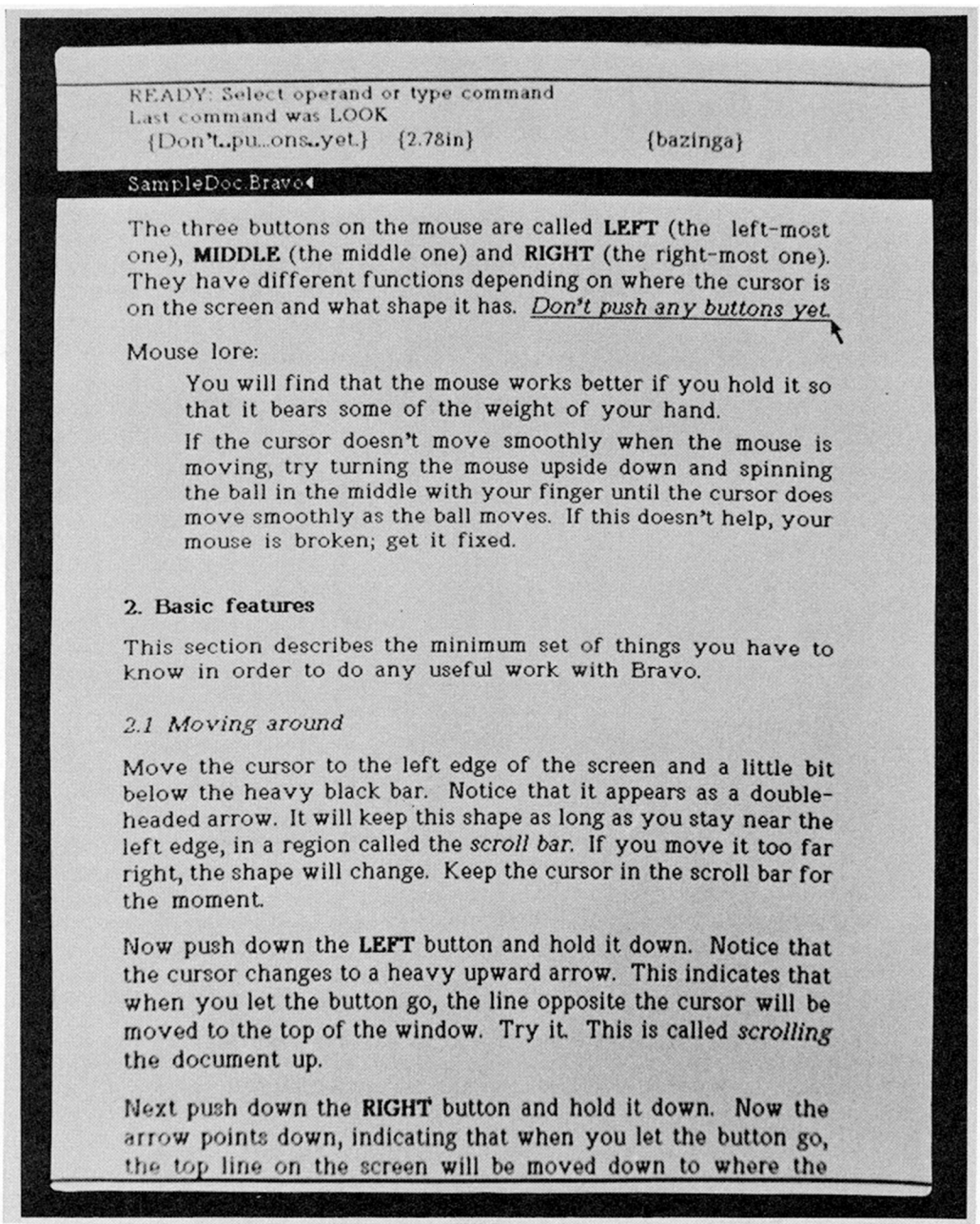

READY: Select operand or type command
Last command was LOOK
{Don't..pu..ons..yet.} {2.78in} {bazinga}
SampleDoc.Bravo
The three buttons on the mouse are called LEFT (the left-most one), MIDDLE (the middle one) and RIGHT (the right-most one). They have different functions depending on where the cursor is on the screen and what shape it has. Don't push any buttons yet.
Mouse lore:
You will find that the mouse works better if you hold it so that it bears some of the weight of your hand.
If the cursor doesn't move smoothly when the mouse is moving, try turning the mouse upside down and spinning the ball in the middle with your finger until the cursor does move smoothly as the ball moves. If this doesn't help, your mouse is broken; get it fixed.
2. Basic features
This section describes the minimum set of things you have to know in order to do any useful work with Bravo.
2.1 Moving around
Move the cursor to the left edge of the screen and a little bit below the heavy black bar. Notice that it appears as a double-headed arrow. It will keep this shape as long as you stay near the left edge, in a region called the scroll bar. If you move it too far right, the shape will change. Keep the cursor in the scroll bar for the moment.
Now push down the LEFT button and hold it down. Notice that the cursor changes to a heavy upward arrow. This indicates that when you let the button go, the line opposite the cursor will be moved to the top of the window. Try it. This is called scrolling the document up.
Next push down the RIGHT button and hold it down. Now the arrow points down, indicating that when you let the button go, the top line on the screen will be moved down to where the

Figure 6. Bravo display. (Courtesy Xerox Corporation.)

ing the thumbnail after the bookmark on the continuum, the user can similarly travel through the remainder of the document.

Bravo uses a postfix/infix interaction syntax: a selection is followed by the *command*, followed by an optional *argument*. For example, deletion works by selecting the scope and pressing the D key, while insertion works by selecting the scope, pressing the I key, typing the desired text, and finally pressing the ESC key.

Selection operates on four main elements: characters, words, lines, and paragraphs. The left and middle buttons of the

mouse are used to select items, while the right button is used to extend those selections. With the cursor in the text area, the left button would cause the addressed character to be selected as the scope, while the middle button would cause the addressed word to be selected. To select the large elements, the user moves the cursor into the line bar. In the line bar, the left button selects a line and the middle button selects a paragraph. Extending the selection allows the user to specify a scope that lies between two of the entities addressed. Thus, clicking left in the text area would cause a single character to be selected; clicking right at some other character would cause all the text between (and including) the two selected characters to be selected. Similarly, a middle right sequence would select all the text between and including two words. In the line bar, a left right sequence selects all the text between and including two lines; a middle right sequence selects all the text between and including two paragraphs.

Operations are typically performed on the current selection. To delete a word, the user simply selects a word by clicking the middle button and then types D to execute the delete command. Similarly, to delete all the text between and including two paragraphs, the user clicks middle right in the line bar and types D. Changes are done analogously. To replace a word, the user clicks middle and types R. Bravo deletes the selected word and puts the user into insert mode; everything the user types until the ESC key is pressed is inserted in place of the old word. The Append and Insert commands allow the user to add text in a similar manner without first deleting a selection. Bravo supplies an undo facility that undoes only the last operation.

Files are never saved until the user explicitly saves them. However, Bravo keeps a transcript of the operations that have occurred in the editing session. One can run BravoBug with the transcript against the old version of the file to interactively replay the editing session. The user is given the choice of single-stepping through each change or running the entire transcript, stopping whenever desired.

At the time of its introduction, the most innovative features of Bravo were its interactive formatting facilities. Bravo's unit for specifying the formatting attributes of text is the *look*. Each character in the document has associated with it particular looks; the looks of any character can be displayed by selecting that character and typing L ?. The looks specify a large assortment of type attributes: font style, point size, subscripting, superscripting, centering, justification, nested indenting, and leading (interline spacing), to name a few, are attributes that the user can change by typing L, followed by a one-character operand. Other look attributes cannot be changed directly by command but are constrained by previous formatting attributes. As soon as a look command is executed, the document is dynamically reformatted to effect the revision—the document is up-to-date in both format **and** content at all times. A special page format mode allows the user to see the document paginated as it will be printed.

Bravo does not allow integrated graphics, but provides output that can be postprocessed to add pictures from the PARC interactive picture-editing systems [BAUD78, NEWM78, BOWM81].

1 4.2 Xerox Star

Star [SEYJ81, XERO82, SMIT82], Xerox's commercial successor to the Bravo, is, in terms of its user interface, the most advanced commercial product for office automation on the market at the time of this writing.

Like the Alto, the 8010 work station on which Star runs is a personal computer with access to shared resources such as file- and printer-servers via an Ethernet network. It has a "landscape" $13\frac{1}{2} \times 10\frac{1}{2}$ inch screen with a resolution of 1024×809 pixels, capable of displaying both a full page of a document and a large menu area.

Several design goals are important to the understanding of Star's interface and functionality:

- The designers determined that users should simply point to specify the task they want to invoke, rather than remember commands and type key sequences. They believed that the user should not need to remember anything (of consequence) to use the system.

- An important consideration was the development of an orthogonal set of commands across all user domains; the copy command in the text formatter, for example, should have similar semantics to one in the statistical graphing package.
- The system was designed to operate by "progressive disclosure." Star strives to present the user with only those command choices that are reasonable at any given juncture.
- Finally, Star is an interactive editor/typesetter; the screen is, for the most part, a facsimile of what the final document will look like.

The Star development team, which worked several years considering possible models, remarks:

> The designer of a computer system can choose to pursue familiar analogies and metaphors or to introduce entirely new functions requiring new approaches. Each option has advantages and disadvantages. We decided to create electronic counterparts to the physical objects in an office: paper, folders, file cabinets, mail boxes, and so on—an electronic metaphor for the office. We hoped this would make the electronic "world" seem more familiar, less alien, and require less training (Our initial experiences with users have confirmed this.) We further decided to make the electronic analogues be *concrete objects*. Documents would be more than file names on a disk; they would also be represented by pictures on the display screen [From "Designing the Star user interface" by D. C. Smith, C Irby, R. Kimball, B. Verplank, and E. Harslem in April 1982 issue of BYTE magazine, © 1982 Byte Publications, Inc. Used with permission of Byte Publications, Inc.]

The high-level conceptual model of the environment is that of a desk top on which multiple documents can be manipulated simultaneously. Star uses a two-button mouse and a postfix interaction syntax. Rather than presenting the user with a simple textual menu or list of available options and files, Star presents graphical icons that resemble the entity to which the user is referring (see Figure 7).

To open a file for editing, the user simply points to the iconic file drawer that symbolically holds the document (noun selection) and issues the open command (verb selection). Choosing open causes a file drawer directory, containing identifiers for file folders and individual documents, to fill part of the screen. The user can open, copy, move or delete any of these folders or documents; touching copy, for example, causes a new document icon to be placed on the user's "desk top" area on the screen. Selecting this icon and opening it causes the editor to open a window that is large enough to hold a facsimile of an $8\frac{1}{2} \times 11$-inch page (see Figure 8). Editing operations similar to those provided by Bravo can be performed in this window. The interface, however, does not use control characters. Mouse buttons and function keys provide the most frequently used commands; a menu of window-specific commands appears in the window banner at the top of the window for selecting with the cursor, and a menu of infrequently used system commands is available by selecting a menu icon in the upper right corner of the screen.

Traveling buttons located on the bottom and right borders of each window, as shown in Figure 8, are selected with the mouse. The $\vdash$ on the bottom makes sure that the left margin of the document is in view while the $\dashv$ makes sure the right half of the document is in view. The $\rightarrow$ scrolls the document to the right, the $\leftarrow$ scrolls the document to the left, the $\downarrow$ scrolls the document downward, and the $\uparrow$ scrolls the document upward. P goes to the previous document page, while N goes to the next document page.

Other icons include a printer icon, a floppy disk icon, and an in/out box icon. To print a file one simply selects the appropriate document icon and places it on top of the printer icon. The programmable cursor changes to an hourglass to indicate that processing is taking place. Similarly, electronic mail is sent by placing a document icon on the out box and is received by selecting the in box.

As in Bravo, the mouse is used to drive the cursor and select elements. Selections are performed with the left mouse button and can be adjusted with the right mouse button. To select a character in the text, the user clicks the left button. Subsequently, when the right mouse button is held down, all the characters between the selected point and the current position of the cursor will be highlighted in reverse video; when the right button is released, the

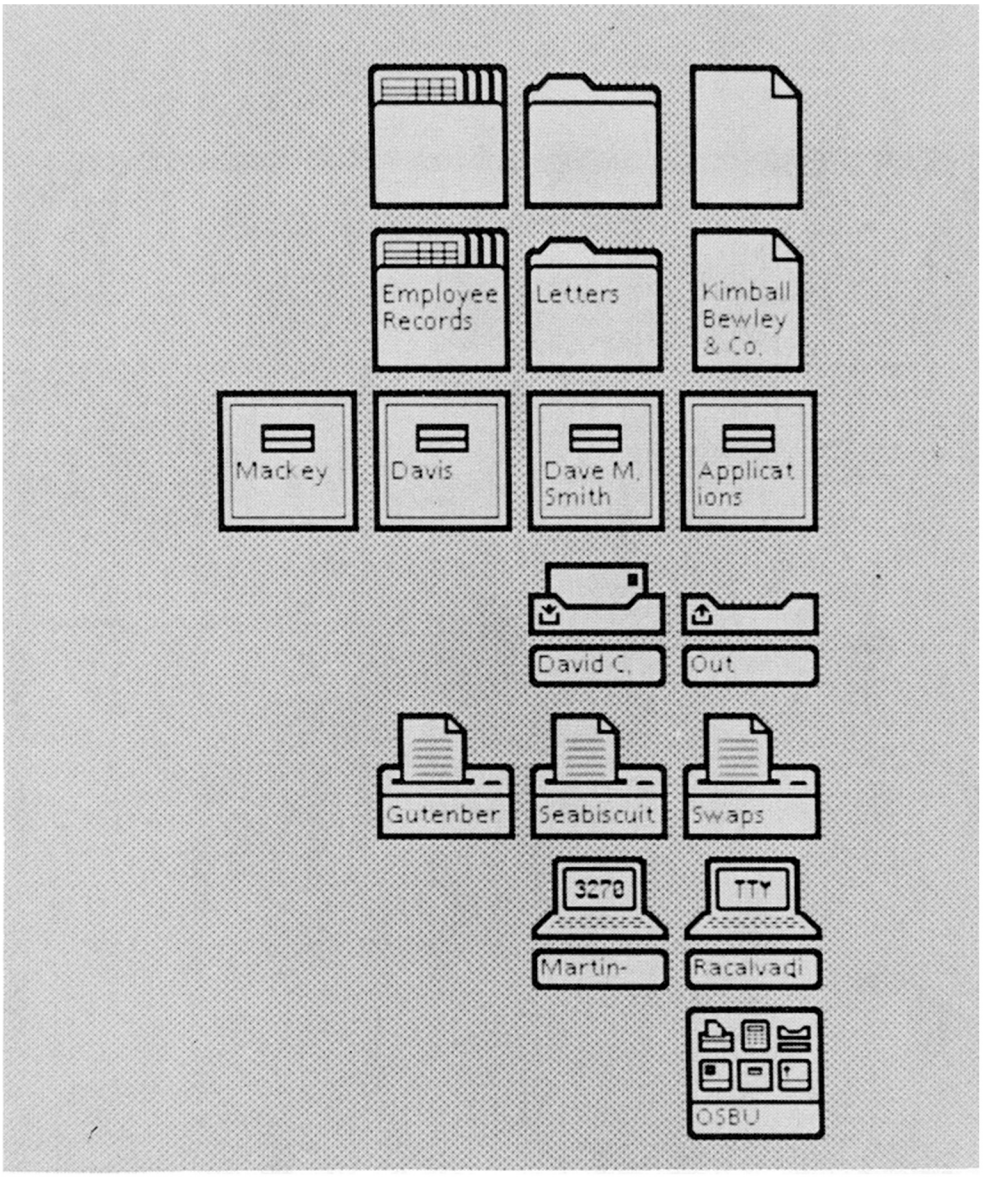
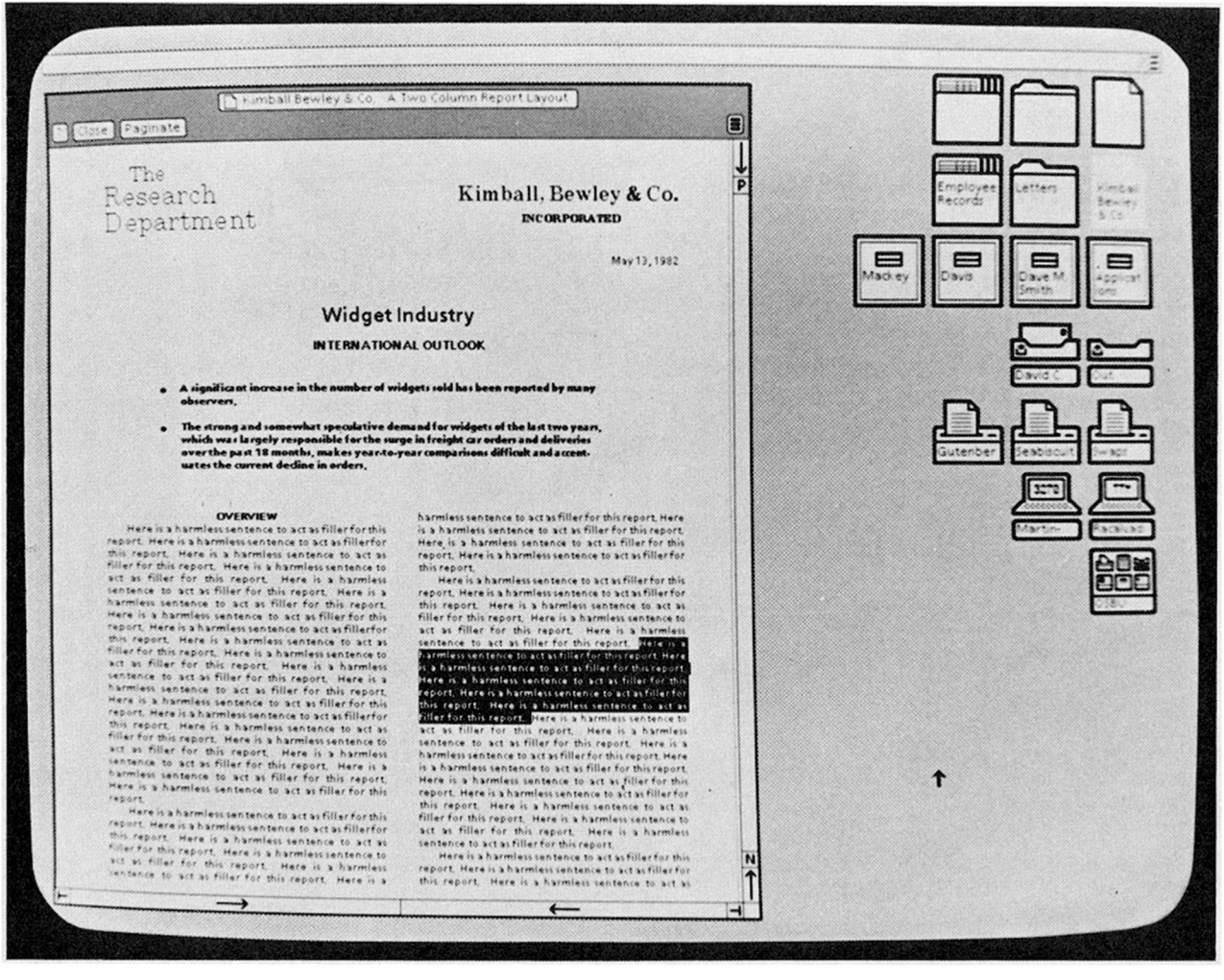


Figure 7. Star icons. (Courtesy Xerox Corporation.)

highlighted area becomes the selection. Two left button clicks select the word containing the cursor; holding down the right mouse button extends the selection to include full words between the selection points and the cursor. Three left button clicks select the sentence containing the cursor; holding down the right mouse button extends this selection by sentences. Four left mouse button clicks select the paragraph containing the cursor; holding down the right mouse button extends the selection by paragraphs. A fifth left button click returns to the original character selection.

Most commands are postfix, requiring simply the selection of an icon or a region of text followed by the issuance of a command using a function key or menu selection. Commands such as find, move, and


Kimball Bewley & Co. - A Two Column Report Layout
Close
Paginate
The Research Department
Kimball, Bewley & Co.
INCORPORATED
May 13, 1982
Widget Industry
INTERNATIONAL OUTLOOK
• A significant increase in the number of widgets sold has been reported by many observers.
• The strong and somewhat speculative demand for widgets of the last two years, which was largely responsible for the surge in freight car orders and deliveries over the past 18 months, makes year-to-year comparisons difficult and accentuates the current decline in orders.
OVERVIEW
Here is a harmless sentence to act as filler for this report. Here is a harmless sentence to act as filler for this report. Here is a harmless sentence to act as filler for this report.

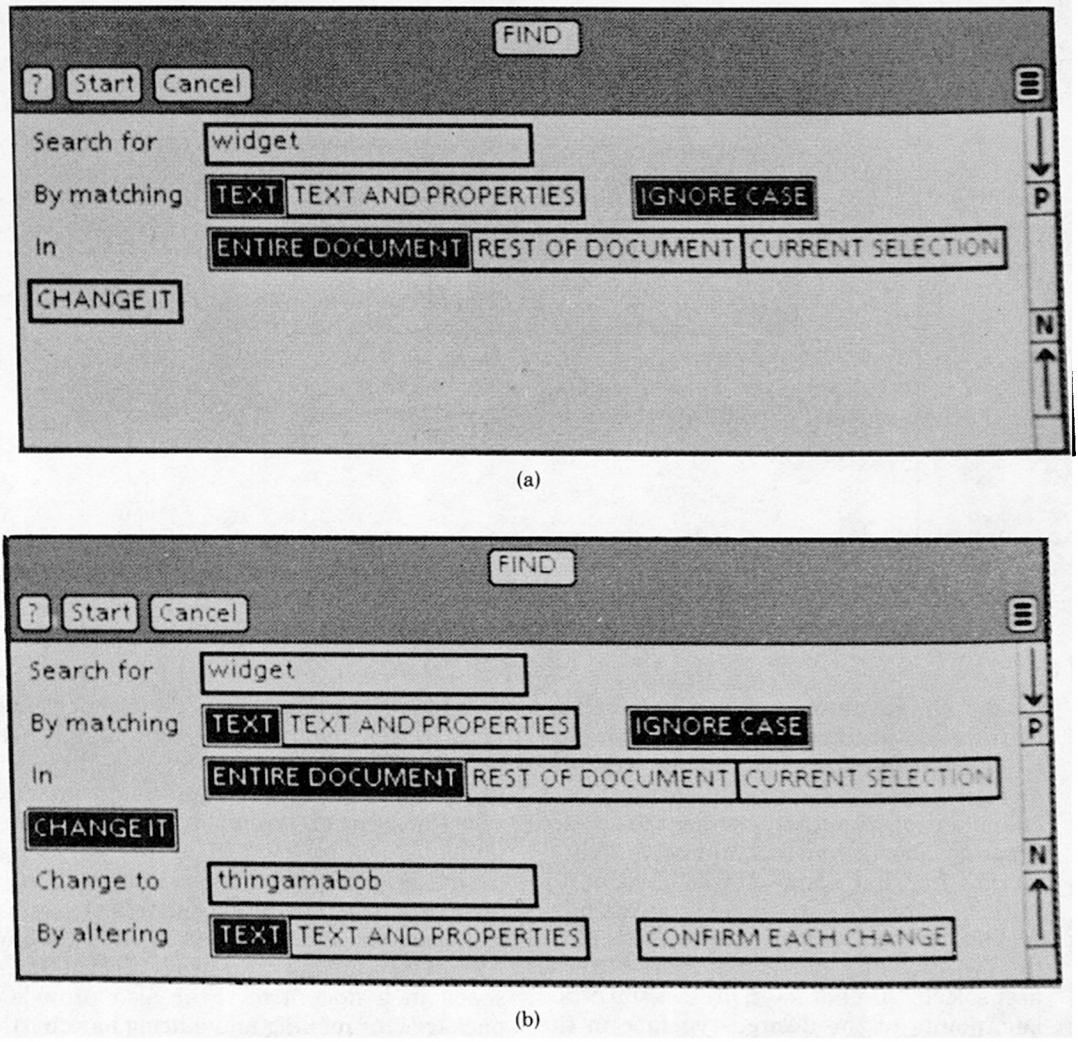


Figure 9. (a) Search operation; (b) search and replace operation. (Courtesy Xerox Corporation.)

copy that need multiple operands are specified in infix or prefix, as appropriate. To perform a find, the user presses the find function key and is given a find property sheet to fill out, as shown in Figure 9a. The user fills in the Search for box by typing a search pattern and specifies other attributes of the search by selecting various options on the property sheet with the mouse. (Here TEXT, IGNORE CASE, and ENTIRE DOCUMENT are selected.) The options that are selected remain selected from search to search until the user explicitly alters them. To perform the search, the user selects the Start button in the window menu. While Star is searching, it displays the message "Searching ..." as feedback for the user. The ? and Cancel button provide help and abort the search, respectively.

The search and replace operation uses the same property sheet. Picking the CHANGE IT button on the find property sheet brings into view a second set of properties. The user can now type in the pattern to Change to and specify what should be altered and whether the replacement should be done with confirmation. When performing these operations, the message "Substituting ..." provides needed feedback.

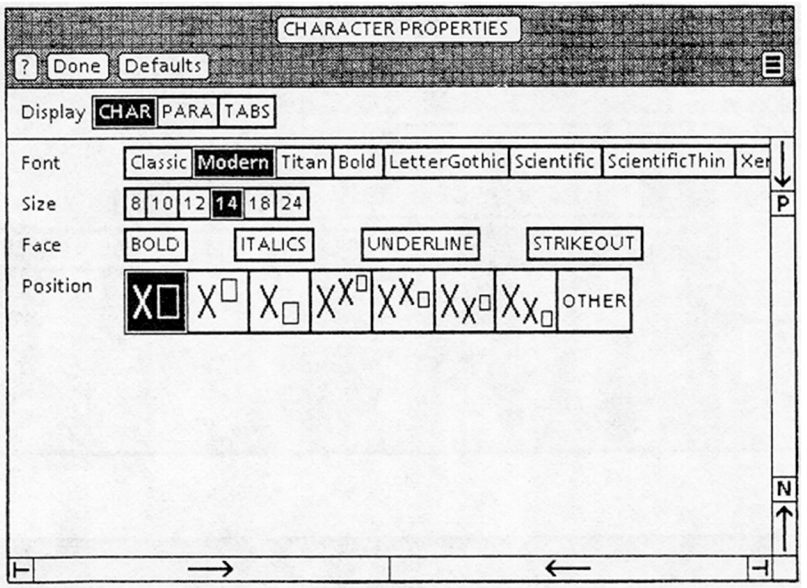


Figure 10. Character property sheets. (Courtesy Xerox Corporation.)

Like Bravo, Star performs instant formatting and justification in the proper type size and style. Associated with each element (an element being any entity from a character to the entire document itself) are property sheets that contain status information for that element (see Figure 10). Initially, attributes in property sheets have system-assigned default values. To change the typeface of a particular character, the user selects the character, presses the props key, points to the desired typeface in the property sheet and closes it. The change takes effect immediately. (Note that the property sheet is presented in the same kind of window as a normal document. In fact, to see all the available typefaces in this example, the user would have to scroll the document to the left with one of the scroll symbols.) Star enables the user to define a standard collection of property sheets to provide document templates (style sheets), as in a database-driven formatter such as Scribe. The user simply copies the template and enters the new text, assured that the basic format is properly defined. The designers compare this to tearing off a standard form from a preprinted pad [SMIT82].

Star provides a drawing package, the results of which can be integrated into a document (see Figure 11). The user selects lines, boxes, shading patterns, and other primitives from a menu and uses these to draw on a user-determined grid. Just as selections can be extended, several graphic items can be selected at once by holding down the appropriate mouse buttons. Users are also allowed to define clusters of graphical items to form new "primitives." Graphics can be scaled up or down to fit in a fixed space in a document. Star also provides packages for making and editing bar charts and spread sheets, and for retrieving information through a relational database system. The designers have stressed what we believe to be a vitally important concern: that all the packages have consistent interfaces, as users especially want a particular command to behave in a consistent way in the provided multiview environment. Whether in the text editor, the graphics editor, or the chart maker, the user issues commands by selecting the object of the operation and issuing the appropriate command through function keys or menu buttons. To delete a word, one selects the word and presses the delete key; to delete a rectangle, one selects the rectangle and presses the delete key.

Besides its carefully crafted user interface, Star provides some interesting solu-

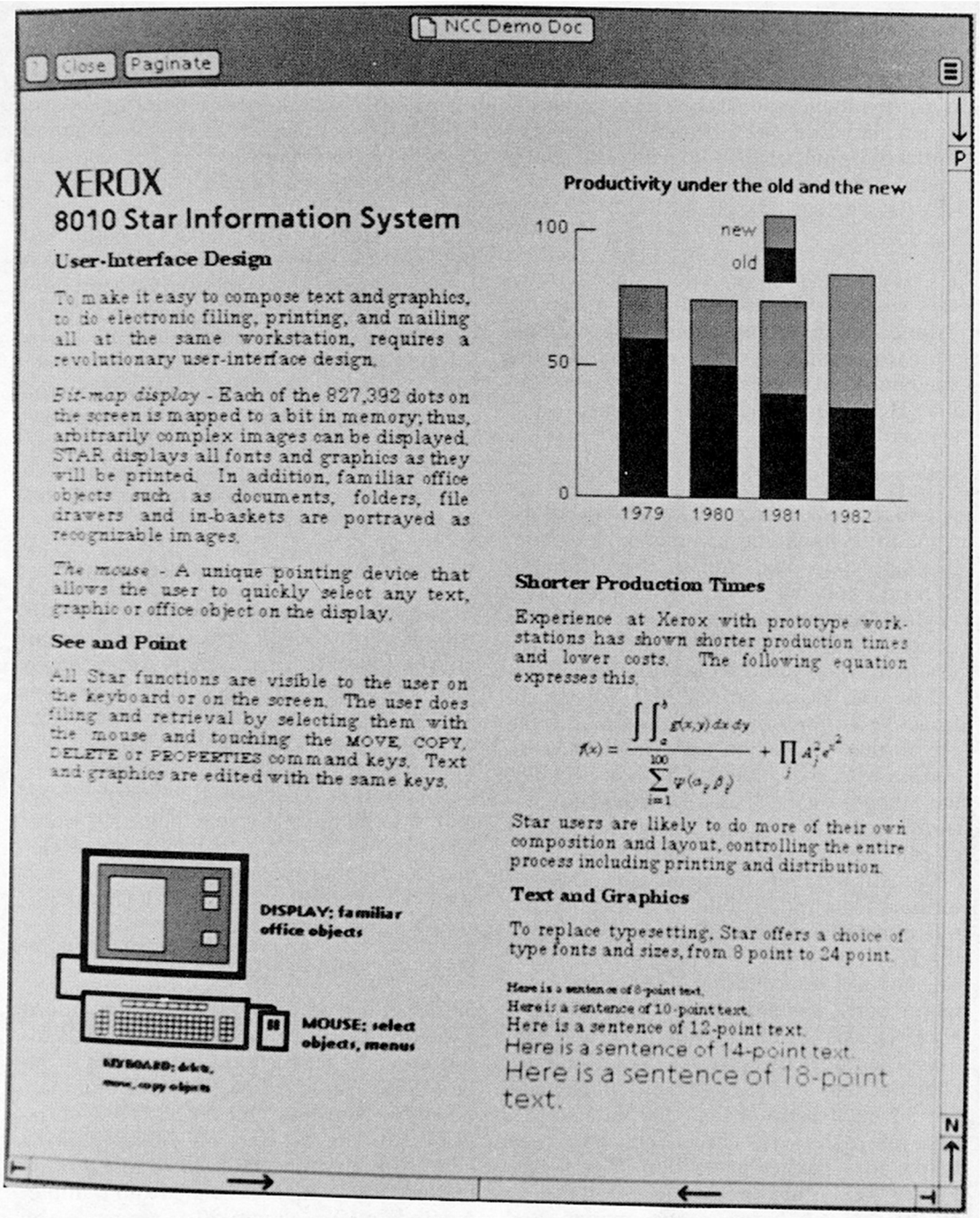

NCC Demo Doc
Close
Paginate
XEROX
8010 Star Information System
User-Interface Design
To make it easy to compose text and graphics, to do electronic filing, printing, and mailing all at the same workstation, requires a revolutionary user-interface design.
Bit-map display - Each of the 827,392 dots on the screen is mapped to a bit in memory; thus, arbitrarily complex images can be displayed. STAR displays all fonts and graphics as they will be printed. In addition, familiar office objects such as documents, folders, file drawers and in-baskets are portrayed as recognizable images.
The mouse - A unique pointing device that allows the user to quickly select any text, graphic or office object on the display.
See and Point
All Star functions are visible to the user on the keyboard or on the screen. The user does filing and retrieval by selecting them with the mouse and touching the MOVE, COPY, DELETE or PROPERTIES command keys. Text and graphics are edited with the same keys.
DISPLAY: familiar office objects
MOUSE: select objects, menus
Productivity under the old and the new
100
50
0
new
old
1979
1980
1981
1982
Shorter Production Times
Experience at Xerox with prototype workstations has shown shorter production times and lower costs. The following equation expresses this.
Star users are likely to do more of their own composition and layout, controlling the entire process including printing and distribution.
Text and Graphics
To replace typesetting, Star offers a choice of type fonts and sizes, from 8 point to 24 point.
Here is a sentence of 8-point text.
Here is a sentence of 10-point text.
Here is a sentence of 12-point text.
Here is a sentence of 14-point text.
Here is a sentence of 18-point text.
P
N

Figure 11. Star graphics. (Courtesy Xerox Corporation.)

tions to typical online manuscript-preparation problems. Mathematical and foreign-language typesetting in most systems involves using escape/control sequences or long English-language mnemonics to represent the special characters. Star presents the *virtual keyboard*, a graphical representation of the keyboard on the screen. To use a key, one simply points at it or presses the corresponding physical key. Star has

knowledge of mathematical symbols and can construct complex equations and formulas as they are typed, changing the size of the symbols used as equations get larger or smaller. Editing continually adjusts the horizontal and vertical spacing and placement of subscripts and superscripts.

1.4.3 ETUDE

ETUDE [HAMM81] is a document production system designed with twin goals: "to extend the functionality of conventional word processing systems while **reducing** the complexity of the user interface." Unlike Bravo or Star, ETUDE uses prefix syntax:

action modifier element

where an *action* might be move or delete, a modifier might be a number or a word like start-of or next, and an element might be paragraph, word, document. The designers feel that the prefix syntax, as in "delete 3 words," more closely approximates natural language, and thus is preferable.

Like Bravo and Star, ETUDE is an interactive editor/formatter, providing typeset, formatted text on a stand-alone work station with a bit-mapped screen. ETUDE has adopted a Scribe-like method for describing formatting, switching the burden of complex formatting from the user to a document database that contains standard formats for a range of documents and document components. In a letter, for example, the ETUDE system will do special formatting for the returnaddress, address, salutation, body, and signature. While ETUDE always keeps the up-to-date formatted document on the screen, it uses the left margin of the screen as a *format window* to place formatting descriptor tags that indicate the type of high-level action that has been taken on a particular section of text (see Figure 12). This technique attempts to bridge the gap between the unformatted but explicitly expressed formatting code, and the displayed facsimile page that, once formatted, often does not contain information about the act that caused the formatting to occur. (A more detailed discussion of the interactive versus batch formatting question is presented in Section 5.)

The user interface is designed for various levels of expertise. The user can call a menu to the screen at any time and select a command with cursor keys or a pointing device. Alternatively, the user can type a command to perform the same action—or use specialized function keys provided for the most widely used commands.

The system provides a cancel command to abort the current operation, an again command to execute the current command again, and an indefinitely deep undo facility. The same tree structure that keeps track of the undo history is used in a help command that creates windows to show the user the session's history and what options are currently available. When the help command is invoked, the user is presented with descriptions of a few past operations plus what is currently being done.

1.5 General-Purpose Structure Editors

Structure editing, pioneered by Englebart with NLS, has been "rediscovered" as an alternative to standard character-oriented methods of editing. Since most target applications have some innate structure (e.g., manuscripts are composed of chapters, sections, paragraphs), the philosophy of structure editors is to exploit this "natural" ordering to simplify editing. The most common representation is a hierarchy of elements. Standard operations on this tree structure, as taken from XS-1 [BURK80], are shown in Figure 13.

1.5.1 NLS/AUGMENT

NLS was a product of research at Stanford Research Institute (now renamed SRI, International) between the early 1960s and late 1970s. Renamed AUGMENT and marketed by Tymshare, Inc., NLS is one of the seminal efforts in the field of text editing and office automation; indeed, many of its features are being reexamined and reimplemented today—almost 20 years since the inception of the NLS project. For example, NLS introduced the notion of conceptual models for the editing and authoring processes, (tree-) structured editing, element modifiers for the editing and viewing operations, device-independent interaction syntax, the mouse as a cursor manipulation

4/18 10:10 RT=85 M=214377 GC=2 L= 0 546 Document: letter:BodyText
Ilitos: [redacted]/Letter

returnaddress

MIT Laboratory for
Computer Science
545 Technology Square
Room 217
Cambridge, MA 02139

March 10, 1980

address

John Jones
World Wide Word Processing Inc.
1378 Royal Avenue
Cupertino, CA 95014

salutation

Dear John:

body, paragrap

We are pleased to hear of your interest in our ETUDE text formatting system, which is now available for demonstration. Enclosed you will find a copy of our working paper entitled *An Interactive Editor and Formatter*, which will give you an overview of some of the goals of our research. This research is funded by a contract with Exxon Enterprises Inc.

paragraph

Our efforts have been guided by a number of general principles:

number, item

1. ETUDE should be easy to use. The system should respond in a reasonable manner, regardless of the user's input. In particular, the user should not be reluctant to try a command, for fear of losing the current document.

item

2. A user of ETUDE should not be concerned with the details of a document's formatting

Figure 12. ETUDE screen. (Courtesy M I.T. Laboratory for Computing Science)

Operation	Effect on Tree
INSERT	Insert a new site (with empty data collection) into a specified gap
DELETE	Delete a subtree (nodes and data)
COPY	Copy a subtree to a specified gap
MOVE	Move a subtree to a specified gap
SPLIT	Split a node and its data into two
MERGE	Merge two nodes and their data
EXPAND	Insert an intermediate level in the tree
SHRINK	Delete an intermediate level of the tree
ORDER	Permute the nodes on a tree level

Figure 13. Tree editor functions of a structure editor. (Adapted from BURK80.)

device, sophisticated browsing and viewing mechanisms, intermixed text and graphics, and even multiperson, distributed editing. At a spectacular, landmark demonstration of the system at the 1968 Fall Joint Computer Conference in San Francisco, text, graphics, and live video of Douglas Engelbart in San Francisco and his colleagues 20 miles away in Menlo Park were superimposed on multiple viewports on the (video projected) screen, as they were working together and explaining what they were doing. "Chalk-passing" protocols were demonstrated for synchronizing multiple users. This demonstration was a forerunner of graphics- and sound-based teleconferencing.

NLS/AUGMENT clearly embodies much more than just a text editor. Its aim is to provide a new way of thinking and working by utilizing the power of the computer in all aspects of one's work:

> We are concentrating fully upon reaching the point where we can do all of our work on line—placing in computer store all of our specifications, plans, designs, programs, documentation, reports, memos, bibliography and reference notes, etc., and doing all of our scratch work, planning, designing, debugging, etc. and a good deal of our intercommunication, via consoles [ENGE68, p. 396 Reprinted by permission AFIPS Press]

Regardless of the subject matter, all NLS information is stored in a hierarchical outline structure of the form

```
1 . . .
  1a . . .
  1b . .
    1b1 . . .
      1b1a . . .
    1b2 . . .
    1b3 . . .
    1b4 . . .
    1b5 . . .
2  . .
  2a . . .
3 . .
4 . .
  4a . . .
    4a1 . .
    4a2 . . .
```

Statements can be nested an arbitrary number of levels. Each statement has associated with it a statement number of the form shown above; these are the main means of referencing the statements from other parts of the text. One statement may be a *substatement* of another statement (1a1 is a substatement of 1a), one may be the *source* of another (1a is the source of 1a1), one may be the *predecessor* of another (4a1 is the predecessor of 4a2), or one may be the *successor* of another (4a2 is the successor of 4a1). NLS provides modifiers to reference not only text elements but structure elements as well. A *statement* is a text node of up to 2000 characters. A *branch* is a statement and all its substatements. A *plex* is a branch plus all the other branches with the same source. The plex of 4a1 is 4a1 and 4a2; the plex of 4a2 is the same. A *group* is a subset of a plex; it consists of all the branches of a plex that lie between and include two branches. The group of 1b2 and 1b4 includes 1b2, 1b3, and 1b4.

The hierarchy is useful for programs as well as for documents since it can be used to model the block structure of the program. Viewspecs allow levels of detail in the outline structure to be made invisible; the viewspecs effect *information hiding*, the selective display or nondisplay of existing material based on attributes provided by the user.

NLS/AUGMENT allows the user to create a hypertext by superimposing on the structure a network of links that point to various discrete statements in this document. In general, these links are specified by the identifier

⟨host, owner, file, statement⟩

which allows the linking of documents over multiple computers.

Commands in NLS/AUGMENT can be executed by using a mouse to select from a menu on the screen, by using the keyboard, or by using both the keyset (described in Part I, Section 2) with one hand to enter the command, and the mouse with the other to make a selection.

The editing commands are quite extensive, providing the first attempt at an orthogonal command syntax with element modifiers. For instance, the insert com-

mand can be modified, with nouns such as word, sentence, and branch. As in most structure editors, the commands are divided into those that operate on the structure (such as move) and those that operate on the text. NLS/AUGMENT provides a very large repertoire of both. Most standard tree manipulations, such as locating or deleting the next node or the previous one, locating the first subnode, and rearranging neighboring nodes, are allowed. The move and copy structure commands provide dynamic renumbering of sections and updating of links throughout the document if necessary.

The system provides the ability to embed control codes in special delimiters within the text both for formatting options such as font changes and for traveling information (links, annotations). These codes can be edited like regular text until they are invoked by special commands (a link is not operable until the jump command is invoked). Viewspec parameters allow one to turn off viewing of these special codes as desired.

A journaling facility provides extensive archiving power for past on-line conversations and teleconferences. Tymshare's commercial version of AUGMENT makes use of TYMNET, a transcontinental satellite network, to satisfy one of the original goals of the project: the sharing of knowledge across great distances. In fact, it is not uncommon for someone in New York to compose a document by making several links to an existing document belonging to a colleague in California.

At its time of introduction, NLS was unusual not only in terms of its functionality but also because of the software engineering environment in which it was produced. This environment included compiler compilers, systems implementation languages, and command language interpreters.

1 5.2 Burkhart/Nievergelt Structure Editor

Burkhart and Nievergelt at the Institute for Information in Zurich have designed a family of structure-oriented editors called XS-1 [BURK80]. The designers contend that the basic sets of editing operations, regardless of the target being manipulated, are similar, and that "a universal structure defined on all data within a system" exploits that similarity to its greatest advantage. As in NLS, the structure of data of all types in XS-1 is represented as a tree, with the nodes ("sites") representing subsets of data. Like many structure editors of its kind, the core of the XS-1 system is a flexible tree editor that allows the user to manipulate the elements at the site (node) level. Fundamental to the XS-1 philosophy is the belief that the user works only on a restricted set of data and with a restricted set of commands at any one time. Therefore, the system supports progressive disclosure, explicitly showing the user the valid command repertoire and operation targets at any given moment. The user always has the familiar tree operations available at all times.

XS-1 provides the user with standard structure editor methods of travel through the explore command. Here, the user can use relative motion to traverse up, down, left, or right in the tree. As well, absolute motion allows the user to move explicitly to something by specification of an identifier such as a name.

The tree editor follows several basic principles. After the completion of any operation, the integrity of the tree structure is guaranteed. (This may be accomplished by attaching target-specific syntax rules to operations, making a syntax-directed editor.) XS-1 provides the ability to specify different views of the same targets, such as a tree structure of a program or an indented view of the same program.

An important aspect of XS-1 is the combination of the same target-independent tree editor with target-dependent back ends to create multiple editors. One is a document editing/formatting system. Here, the author sees on the screen a rectangular window into the text and a text cursor. All high-level operations (move, copy, etc.) are handled by the target-independent tree editor; only a small set of text editing primitives at the character, word, or sentence levels is provided. The command set is consistent between targets; operations provided by the universal editor are also pro-

vided for specific target-dependent modes, enabling the user to deal with a relatively small set of operations that do "obvious" things. For example, a move command in a text editor would move the selected text from source to destination, while a move command in a graphics editor would move the selected graphics object from source to destination. Text formatting is done by appending a formatting descriptor to each site; these can be edited by the tree editor as well.

1.5.3 Fraser's s

Fraser's *s* [FRAS80] is an attempt to provide standard editing primitives that can be used to build a variety of editors. *s* allows the programmer quickly to create different front ends for a text editor so that various targets can be modified using existing editing routines.

The philosophy behind *s* is that many computer utilities—interactive debuggers, file system utilities, even tick-tack-toe games—are simply editors in that they accept a particular input syntax and modify the existing representation and/or state of their particular data. Rather than producing languages and scanners for each application, *s* attempts to use a generalized structure and a generalized text editor nucleus for editing all applications.

One application allows the user to edit UNIX *i nodes*, complex (18-field) data structures containing pointers and information about a file block from the UNIX file system. When the system crashes or a disk block becomes unusable, the systems programmer occasionally has to go into the file system and manually change pointer values from a dump-type format. *s* provides a screen-based view of the file descriptor, allowing the user to edit each of the fields, which are represented one per line. An overstrike, for example, is translated into a call to the nucleus routine fetch to retrieve the appropriate field and a call to the nucleus routine change to update the field. The deletion of a field would be performed with a call to the nucleus routine delete.

Another interesting use of *s* is as a UNIX file directory editor. The UNIX ls -l command provides a listing of file attributes:

```
drwxr-xr-x 2 nkm    224 May  9 15:27 bBACKUP
-rw-r--r-- 1 nkm  36585 May  2 16:42 section1.tex
-rwxrwxrwx 1 nkm  16714 Apr 25 17 11 section2 tex
-rw-r--r-- 1 nkm  48414 May  2 16:44 section3.tex
-rw-r--r-- 1 nkm  55282 May  6 00:23 section4a tex
-rw-r--r-- 1 nkm  20113 May  6 00 49 section4a2 tex
-rw-r--r-- 1 nkm   9209 May  9 14 50 section4b.tex
-rw-r--r-- 1 nkm  22049 May  9 14:20 section4c tex
-rw-r--r-- 1 nkm  26958 May  6 02 24 section4d tex
-rw-rw---- 1 nkm   3362 May  9 15 10 section4e tex
-rw-rw---- 1 nkm  18541 May  7 11 13 section5.tex
```

The first field contains a d if that entry is a directory; this field is not editable. The next nine fields contain r, w, and x for read, write, and execute privileges for the owner, the group, and for all others, respectively, with a – indicating no access. The next field, the link count, is not editable. The next field contains the owner of the file. The rest of the fields are not editable, except for the last entry, the actual file name.

Rather than forcing the user to use the UNIX shell commands for performing renaming (mv oldname newname), deleting (rm filename), changing ownership (chown filename), and changing access rights (chmod a+rwx filename to allow all to read, write, and execute the file), the *s* directory editor allows the user to edit the listing directly, barring protected fields. Deleting the characters, r, w, or x removes read, write, and execute access for the corresponding parties; overstriking a – with r, w, or x adds access. Typing over the owner name changes the owner, typing over the file name changes the file name. Deleting an entire line deletes that file.

A different front end allows the user to edit the state of a simple pedagogical computer. Rather than having the student submit punched cards in batch mode and easier and cheaper than having a physical laboratory machine, an *s* front end was written representing the machine architecture as editable lines and allowing the students to modify the appropriate fields. While the goals of the i node and machine applications are different, the primitives to edit them, at least from a system view, are the same.

While the *s* editor was a limited experiment, its ramifications are wide ranging. Many applications, especially ones that are computer based, have some aspect that requires editing. We feel that Fraser's basic

premise—when changing a file name in a file system, when adding a user to a mailing list, or when editing a UNIX *i node* as above, there is no reason why the user should have to resort to special maintenance programs—will be an important goal in the future of editing. As Fraser's *s* has shown, a general-purpose editor can be used to give the user a far more common interface across diverse applications than typically exists today. Moreover, with an appropriate interface, one can perform editing on a graphical representation of the target rather than on an unfamiliar, textual representation.

1.5.4 Walker's Document Editor

Walker's Document Editor is an attempt to design an editor for the preparation of complex documents such as technical manuals. An initial goal of the system was to "develop a structured description for documents ... distinct from any particular commands in the document source" [WALK81b, p. 44]. The Document Editor uses EMACS as a base text editor and Scribe as a document-description database and compiler.

The Document Editor operates on a "document" as a collection of files in Scribe manuscript file form; it infers the structure of the document from the tags in the file being edited. The specialized functions for technical writing provided by the Document Editor are actually extensions to EMACS in the form of a user library.

The Document Editor provides four major categories of document structure editing commands: *locators*, *selectors*, *mutators*, and *constructors*. Locator commands allow the user to specify places in the document; these include commands to go up and down a structural level (e.g., from section to subsection), to go to the next or previous item at the same structural level, and to go to the next structural element of any type. Selector commands allow the user to determine the current makeup of the document by checking the status of the parts and the structure of the document at various (user-specified) levels. Mutators revise the structural makeup of the document, providing functions such as change structural level (e.g., make a chapter a section). Constructors allow the user to create and copy structural elements.

The Document Editor uses Scribe's cross-referencing commands for maintaining cross-references for section numbers, table numbers, and other document information. This facility provides a follow CREF (cross-reference) pointer function to allow the user to view the target of the cross-reference. More interestingly, it contains the find all fingers function, which allows the user to see which cross-reference pointers in the document point to a particular spot in a document. This forms a rudimentary hypertext capability [NELS67, VAND71a], but requires the high computational overhead of being extrapolated from Scribe, rather than being an editor primitive.

The Document Editor uses the cross-reference capabilities to provide functions that manage the task of creating an index for a document. For traveling, the user can follow an index pointer and examine all the fingers pointing to a location, as well as make an index entry, show index symbols, and find all the index symbols containing a particular word.

The Document Editor runs Scribe as an underlying formatting process. The editor itself, EMACS, does not present the formatted text for the user to edit. As discussed in more detail in Section 2, Walker contends that for large documents, one has little interest in anything but the content and the formatting abstractions (as opposed to the actual formatting) during most of the life of the document. However, the Document Editor does provide the functions for compiling those parts of the document that have actually changed, while conforming to the formatting constraints of the entire document (proper page numbers, indentation levels, margins, typefaces). This alleviates the cost of recompiling an entire document because of minor editing changes.

1.6 Syntax-Directed Editors

Syntax-directed editors attempt to increase the productivity of the programmer by removing the time-consuming process of eliminating syntax errors. Syntax editors

are structure editors that ensure that the structure always is constrained to preserve syntactical integrity. Often syntax-directed editors do not merely recognize the syntax and translate the user's actions into linear text, but instead parse the input into an intermediate form that can be used to generate code. Here the editor is both a tool for the programmer and a tool for the compiler/interpreter. We give some prototypical examples below.

1.6.1 Hansen's EMILY

Hansen's EMILY [HANS71] is one of the earliest syntax-directed editors. Rather than typing in arbitrary text, the user creates and modifies text by graphically selecting units of text (*templates*) that are constructs in a programming language. Text is created with a sequence of selections. The screen is divided into three areas: text, menu, and message. The text area in the upper two-thirds of the screen displays the text under construction as a string that contains the nonterminals (nonatomic entities) of the program, highlighted by underlining. The current nonterminal is enclosed in a rectangle. The menu in the lower third of the screen displays a set of possible replacements for the current nonterminal. The user selects a replacement rule and the system makes the substitution, locates a new current nonterminal, and displays a new set of choices. The message area is used for entering identifiers and also displays status and error messages. Assuming a partial PL/I-type grammar like

```
⟨STMT⟩ ::= ⟨VAR⟩ = ⟨EXPR⟩;|
  IF ⟨EXPR⟩ THEN ⟨STMT⟩|
  DO; ⟨STMT*⟩ END;
⟨STMT*⟩ ::= ⟨STMT⟩|⟨STMT⟩ ⟨STMT*⟩
⟨EXPR⟩ ::= ⟨EXPR⟩ + ⟨EXPR⟩|⟨VAR⟩
⟨VAR⟩ ::= id
```

where symbols surrounded by "⟨" and "⟩" are nonterminals, an IF statement might be created in the following manner.

The current (boxed) nonterminal is ⟨STMT⟩, and the menu displays the three choices

```
⟨VAR⟩ = ⟨EXPR⟩;
IF ⟨EXPR⟩ THEN ⟨STMT⟩
DO; ⟨STMT*⟩ END;
```

The user selects the second with a light pen and gets the expansion

```
IF ⟨EXPR⟩ THEN
    ⟨STMT⟩
```

The current nonterminal is now ⟨EXPR⟩, and the menu displays the possible expansions for this. Subsequent derivations to arrive at the appropriate IF clause are

```
IF ⟨VAR⟩ THEN
    ⟨STMT⟩
IF FIRSTTIME THEN
    ⟨STMT⟩
IF FIRSTTIME THEN DO;
    ⟨STMT*⟩
END;
⋮
IF FIRSTTIME THEN DO,
    FIRSTTIME = FALSE;
    SYMBOLS = NULL;
    ENDTIME = DAYMINUTES + 10;
END;
```

Since a syntax imposes a hierarchical structure on text, EMILY can be used for any hierarchical text structure. Each selection from the menu generates a node with space for one pointer for each nonterminal in the replacement string. When a nonterminal is replaced, the corresponding space is filled in with a pointer to the node generated for the replacement. Each nonterminal thus generates a subtree of nodes that is presented on the display, through a tree-walking display routine, as a string of text.

As in NLS, the user can change the view of the text, so that the string generated by any nonterminal is represented by a single identifier called a *holophrast*. For example, the IF statement above could be displayed with all text generated from the ⟨STMT*⟩ represented by a holophrast. In larger programs, this feature means that the user can view the structure of the text without viewing the details. Alternatively, the user can descend into the structure and view the details in full.

Text is also modified in terms of its structure. The text represented by any holophrast can be deleted, moved, or copied. When text is deleted, it is not destroyed immediately, but is automatically moved to

a special system fragment called *DUMP*. If a mistake is discovered before the next text modification is made, the deleted text can be retrieved from this dump.

EMILY is a pure syntax-directed editor. Statements are derived by the menu-picking scenario down to the lowest level, for example, the identifier. This makes the editing awkward, since the user must often traverse long derivations to type in a simple identifier or assignment statement.

1.6.2 Cornell Program Synthesizer

Much work in individual areas was done after EMILY, most notably the MENTOR [DONZ75, DONZ80] tree-manipulation and programming environment, the CAPS diagnostic programming system [WILC76], and the INTERLISP Programmer's Assistant [TEIW77]. The Cornell Program Synthesizer [TEIT81a, TEIT81b], running on both the Terak (LSI-11 based) personal computer and the VAX family of computers, combines many of the ideas from these and other projects into a syntax-directed editor and programming environment for PL/CS, and more recently, PASCAL.

In the Synthesizer, designed for simple terminals which use cursor keys as the only locator device, the user types textual commands that represent the set of possible expansions of the current nonterminal. The set of possible commands can be displayed in an optional window so that the user need not memorize the command sequences. The synthesizer differs markedly from EMILY in that it is not a pure derivational syntax-directed editor. Rather, the synthesizer is a hybrid between the traditional structure editor and the character-string text editor. Thus common elements such as identifiers, expressions, and assignment statements do not have to be considered as elements of a tree structure, nor do they have to be edited and stored as such.

The user is presented with three types of high-level entities. *Templates* are program constructs that need to be filled in. *Place holders* are tags in the template describing the parts that need to be completed, and these are the only parts of templates that can be altered. *Phrases* are pieces of text, not structure, that are typed in to replace place holders.

To start a PL/CS program editing session, the user types •main followed by a carriage return to obtain the template for a PL/CS main program.[4] This template is of the form

```
/* comment */
file-name: PROCEDURE OPTIONS (MAIN);
   {declaration}
   {statement}
   END file-name;
```

The user can position the cursor at the place holder comment and type a phrase containing the text of a comment. Now the user positions the cursor at the place holder for the nonterminal declaration. Since this is a nonterminal (indicated by the braces), the user must select an applicable template for further derivation. At this point, the user can type •fx for a fixed variable, •fl for a float variable, •bt for a bit variable, •ch for a character variable, or •c for a comment. For our example we choose •fx. This expands to the template

```
DECLARE (list-of-variables) FIXED [attributes];
```

The cursor is moved to the list-of-variables place holder, and a phrase containing the name of the variable is typed in. This name, typed in as text, not as structure, is parsed for syntactic correctness upon pressing carriage return, and is stored and manipulated as text. If an illegal variable name is typed, this phrase is highlighted in reverse video and flagged internally. If the attributes are not inserted, the square brackets indicate that default values will be used. The declaration nonterminal is now completely defined, and the user moves on to expand the statement nonterminal, for which there are 13 possible templates. Typing •ie generates the template

```
IF (condition)
   THEN statement
   ELSE statement
```

(The box here indicates the current cursor position.) Typing •p at this position gener-

[4] •, long, clip, delete, left, right, up, down, and diagonal are function keys on the synthesizer keyboard.

ates the PUT template, giving

```
IF (condition)
   THEN PUT SKIP LIST (list-of-expressions);
   ELSE statement
```

The user could then type a phrase like " 'min = ',beta " to fill in the place holder.

The user uses the left, right, up, and down cursor keys to traverse the structure. In fact, the key names do not represent the true functions attached to those keys. Right and down both move the cursor forward through the program; left and up move it backward through the program. Rather than moving character by character, these keys move the cursor one program element (template beginning, place holder, or phrase) at a time. Left and right, additionally, stop at each individual character in a phrase. In an expanded template like the one above, the cursor would stop at the underscored places when using up and down:

```
IF (alpha < beta)
   THEN PUT SKIP LIST ('min = ',beta);
   ELSE statement
```

and at these underscored places when using left and right:

```
IF (alpha < beta)
   THEN PUT SKIP LIST ('min = ',beta);
   ELSE statement
```

The two-key sequence long down (up) moves the cursor to the next (previous) structural element of the same level. Other keys move the cursor to the nearest enclosing structure template and to the beginning of the program.

Insertion and deletion are based on the pick, put, and delete buffer concepts. The user positions the cursor at an appropriate template or phrase, and then issues the delete command to delete that template (including, of course, all subtemplates) or phrase and store it in the delete buffer. Similarly, clip will store a copy of the selected entity in the clip buffer, but not delete the original. The insert command allows the reinsertion of the deleted or clipped text at the current cursor position. In the above example, if the cursor were positioned at the P in "PUT," the sequence delete, down, insert would result in the program segment

```
IF (alpha < beta)
   THEN statement
   ELSE PUT SKIP LIST ('min = ', beta);
```

Correcting mistakes can only be done by preserving structural integrity. Assume the following incorrect code segment:

```
/* compute factorials from 1 to 10 nonrecursively */
a = 0;
DO WHILE (a < 10);
   a = a + 1;
   fact = 1;
   PUT SKIP LIST (a,' Factorial =');
   temp = a;
   END;
DO UNTIL (temp = 1);
   fact = fact * temp;
   temp = temp - 1;
   END;
PUT SKIP LIST (fact);
```

The traditional programmer, realizing that the END of the DO-WHILE loop should properly come at the end of all of this code (nesting the DO-UNTIL and the PUT SKIP LIST), would move the END statement to the end of the code with a single move command or a delete/put sequence to achieve

```
/* compute factorials from 1 to 10 nonrecursively */
a = 0;
DO WHILE (a < 10);
   a = a + 1;
   fact = 1;
   PUT SKIP LIST (a,' Factorial =');
   temp = a;
   DO UNTIL (temp = 1);
      fact = fact * temp;
      temp = temp - 1,
      END;
   PUT SKIP LIST (fact);
   END;
```

In a syntax-directed editor, since the END is part of the DO-WHILE template, it cannot be separately moved. Instead of moving the END forwards, the equivalent backward move of the two following statements must be done. To perform the desired alteration, the user would have to position the cursor at the start of the DO-UNTIL template, press long delete, move the cursor to the last element in the list of structures to be moved (the PUT SKIP LIST (fact) state-

ment), signal completion of the selection by typing •, move the cursor to the structural element after which the new part should be inserted (the temp = a; phrase), press carriage return to open a statement place holder, and issue •ins DELETED to position the desired text in the desired spot. While this is certainly more complicated than the traditional method, the interface is partially to blame. A pointing device that would easily allow selection of elements and extension by structural or contiguous units would eliminate many of the keystrokes above. Even without the pointing device, one could imagine extending the starting or ending portions of a template to encompass contiguous statements.

Even if many syntax-directed editing techniques are nominally longer than traditional techniques, the excess time must be weighed against the time saved by ensuring that a program is syntactically correct every step of the way. One major time-wasting operation that is avoided is the back mapping of frequently inscrutable syntax-error messages to the source lines, all too often a heuristic and frustrating process. Indeed, an important contribution of the Synthesizer project was the concept of the syntax-directed editor as an integral part of a programming environment. The Synthesizer is not typically used to create text files that will later be passed to a standard compiler, but rather as an editor that will create a representation of a program suitable for on-line interpretation. The Synthesizer allows the user to run a program and watch the cursor step through the lines of code as they are being executed, much like the "bouncing ball" familiar to cartoon watchers. Information hiding (such as seeing only the comments or top-level templates) still allows single-step viewing of the program in which the cursor jumps from one visible high-level unit to the next; the user does not have to watch the low-level details, for example, the inner workings of a loop. Uninitialized variables are flagged, type checking is enforced interactively, and duplicate declarations are prohibited, all at edit time, rather than at compile time. Invalid phrases are highlighted as soon as the user types them in. A syntax-directed approach avoids the time-consuming back-mapping error messages from a batch compiler to the proper lines in the source file by generating the error messages interactively, with the offending program components highlighted. Programs are incrementally compiled, allowing the user to reedit and experiment with small parts of a program without waiting for an extensive recompilation. In fact, the approach taken with the Synthesizer allows the suspension of program state, the correction and incremental compilation of a portion of the program, and the resumption of the program.

Templates can be input only in a structurally sound manner, while phrases, typed textually, are allowed to be erroneous. When editing, the user does not need to expand all nonterminals or remove all errors in phrases. An incomplete or erroneous program can be run at any time. However, these irregularities are highlighted from the moment they are input until the moment they are corrected; the synthesizer relaxes some of its constraints, but warns the user accordingly. In both cases, the program will run normally until the error or unfinished program construct is encountered. When this is encountered, the user is free to correct or insert the code and continue the execution.

The program is stored as a combination of a parse tree for the templates, and as actual text for the phrases. The pretty-printed code that the user sees is actually an interactively generated view of the internal data structure.

Currently, a Synthesizer Generator is being developed which will allow a complete syntax-directed editor to be generated from a formal description of the syntax. We point the reader to the GANDALF project at Carnegie-Mellon University [HABE79, NOTK79, FEIL80, MEDI81] for a description of a similar syntax-directed editor and editor-generator project.

1.6.3 Fraser's sds

Fraser's *sds* is a general structure editor driven by a grammar that describes a hierarchical data structure. Our interest in it results from the stress that has been put upon imposing a syntax on targets that are not necessarily programs, and upon the

generation of the editor from a procedural description.

The user-viewable part of *sds* is a screen editor which displays a current record of some tree structure. The cursor keys down, up, left, right and home allow the user to move down to a node field, back up, left or right to adjacent fields, or to the root of the structure. Other commands consist of typing a period followed by the name of a nonterminal, the technique used in the Cornell Program Synthesizer. This causes the editor to allow the user to enter the first field of this new nonterminal. The user can either enter another nonterminal designation or, if applicable, simply type a string that will become a terminal or leaf node. As well, *sds* provides target-independent commands such as .w (.r), which write (read) a subtree to (from) a file, .hide(.show), to suppress (exhibit) detail of a subtree, .pick, which saves a pointer to the current node, and .put, which substitutes the current node with the previously picked node.

The target-specific editor is written using a formal syntax description similar to that used for the YACC compiler-compiler of Johnson [JOHN75]. The entire grammar for an *sds* binary tree editor is captured in one line:[5]

```
tree = value tree tree : dotree(value, tree, tree2)
```

The phrase before the colon is the grammatical description of tree, the only production in the grammar of binary trees. The portion after the colon is SNOBOL4 code to perform an action (tree2 is the second argument named tree, tree*n* would be the SNOBOL argument for the *n*th tree token in a production list). In this example, the dotree subroutine contains SNOBOL code to display the value and the two subtrees in graphical form. Note that to change the representation of a binary tree node to one in which the value lay between the two tree pointers, one would simply have to change the production to transpose the words "value" and "tree":

```
tree = tree value tree : dotree(value, tree, tree2)
```

Similarly, the dotree routine could be changed to store the binary tree in a disk-oriented form or to print it in an indented representation; the actions are independent of the creation routines of *sds*.

A document editor has also been written in *sds*. Of course, this implies the construction of a hierarchical grammar for a document, coupled with action rules for each production. A sample grammar for a small document system looks like

```
paper = title sect : center(title) nl nl
                     generate(sect)
sect = header pp sect : header nl nl put(pp)
                        generate(sect)
pp = text pp : break(text) nl generate(pp)
```

To the right of the colons lie production-specific SNOBOL code. We are concerned here only with the productions to the left of the colon.

To use this editor, the user would enter textual commands to create various levels of the subtree as follows. The prompt line gives the user an idea of location in the structure. The last item on the line is the current field (item on the right side of the production), while the preceding items are the types (items on the left side of the production) which brought the user to that field, that is, the successive nodes of the tree branch. The root name paper is implied at the beginning of each line.

First, the user types .paper, telling *sds* to begin a node of type paper, the root of the structure:

prompt:
user: .paper

The next prompt asks the user to type in a title and go to the next part of the production:

prompt: title
user: Interactive Editing Systems

sds is now ready to perform the sect production, but requires the user to issue the explicit command to create the section:

prompt: sect
user: .sect

Having created the .sect record, the system prompts the user to fill in the header field:

prompt: sect header
user: Introduction

[5] Examples are adapted from FRAS81.

The user is now prompted to create a .pp record, and again must issue an explicit command:

prompt: sect pp
user: .pp

The user is prompted to enter text. In this mode, he is provided with a target-dependent text editor based on the Irons model:

prompt: sect pp text
user: The interactive editor has become an essential...

Upon terminating the paragraph, the user is prompted to create another, as the pp production is recursive:

prompt: sect pp pp
user: .pp

The user then types in the appropriate text:

prompt: sect pp text
user: Though the editor has always been deemed...

The command *up* goes up one level in the structure. This causes a production (pp = text pp) to be completed and an action to be performed, in this case, formatting of a paragraph:

prompt: sect pp pp pp
user: up

We go up one more level of the tree, formatting the first paragraph.

prompt: sect pp pp
user: up

While the user is entering text, *sds* is performing syntax checking, flagging and prohibiting invalid structure at any point in the document.

Initial reaction to document creation by structure centers on the apparent "wordiness" necessary to get the job done, but Fraser contends that the explicit structure is almost identical to what one does implicitly with a compiler-based document language. In fact, the .w command would store the above paper as

```
.paper
Interactive Editing Systems
.sect
Introduction
.pp
The interactive editor has become an essential...
.pp
Though the editor has always been deemed...
```

A third application for *sds* is as a picture editor for simple line drawings. The structure editor, using the small six-line grammar described in FRAS81, would create the multicolor letter "T" with the structure

```
.branch
.color
blue
.line
0,20 20,20
.color
.red
.line
10,0 10,20
```

Other grammars used by *sds* include one for a subset of C [KERN78c].

1.7 Word Processors

1.7.1 WordStar

WordStar [MICR81] is one of the most popular word processing programs available for home computer systems. It runs on a variety of systems under the CP/M operating system, using the CP/M file system to maintain its files.

The conceptual model of text in WordStar is the quarter-plane of the Irons model. Control key combinations (special prefix characters are used as software shift keys to provide a large set of commands), function keys and cursor keys are used for command specification. WordStar combines the quarter-plane model with a "virtual typewriter" model. The user is presented with a ruler line that simulates tab rack and margin ruler on conventional typewriters, and with commands to move virtual margin keys forward and backward on this ruler line. WordStar divides the file into logical pages that default to contain 55 lines (the number of lines on an 8½ × 11-inch page, excluding margins). Most importantly, WordStar provides modest interactive editor/formatter capabilities for justified, monospaced text. As the user types in text, the lines are automatically justified. When text is changed, rejustification is not automatic, but is done on a per-paragraph or per-document basis by user command.

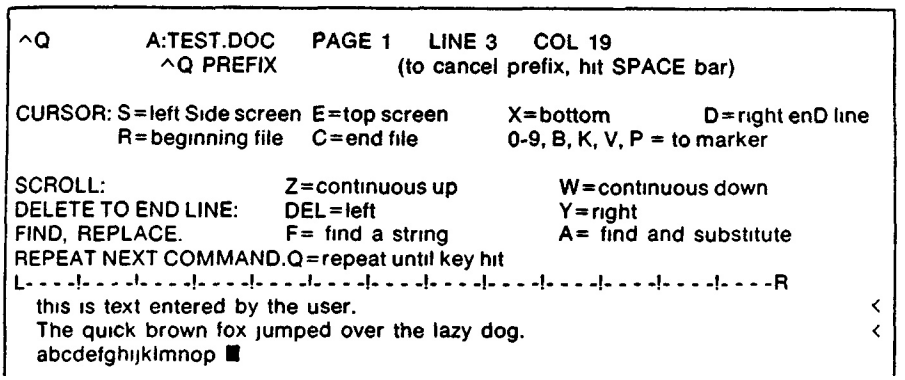

^Q A:TEST.DOC PAGE 1 LINE 3 COL 19
^Q PREFIX (to cancel prefix, hit SPACE bar)
CURSOR: S=left Side screen E=top screen X=bottom D=right enD line
R=beginning file C=end file 0-9, B, K, V, P = to marker
SCROLL: Z=continuous up W=continuous down
DELETE TO END LINE: DEL=left Y=right
FIND, REPLACE. F= find a string A= find and substitute
REPEAT NEXT COMMAND.Q=repeat until key hit
this is text entered by the user.
The quick brown fox jumped over the lazy dog.
abcdefghijklmnop

Figure 14. WordStar screen. (From MICR81. Reprinted with permission.)

The screen is set up to provide extensive feedback to the user. The first line is a status line: it presents the file name of the document, the current page number, and the current line and column at which the cursor points; as soon as the cursor is moved, the latter values are changed. As well, the beginning of this line is used to echo the typed command. For instance, as in Figure 14, if the user types CTRL-Q on the keyboard, the textual representation ^Q is shown on the screen. The next few lines on the screen (above the ruler line) represent the current options. Here, since CTRL-Q was typed, the ^Q prefix options are displayed in the help area. The user then chooses one of the ^Q suffixes, which represent commands. A more sophisticated user can avoid this extensive prompting in two ways. First, if the entire command, say CTRL-QF is typed together quickly, it is executed without displaying the ^Q options. More explicitly, the user is given commands to change help levels. These help levels range from displays for the novice, containing complete options, to those for the expert, containing no options at all. The full set of WordStar commands is shown in Figure 15. WordStar makes sure that the user has noticed an error by requiring an acknowledgment—by default hitting the ESC key—to resume operation.

As in the Irons model, editing is done on the displayed viewing buffer/editing buffer by driving the cursor around and typing. WordStar offers both insert mode and typeover mode.

A major flaw of WordStar is the lack of an undo facility: once a command is executed, it cannot be reverted. This reduces the freedom of experimentation that an author should have. The only recourse that a user has is "undoing" an entire session with an abort command.

A problem with WordStar, and with most microcomputer editors, is lack of both main memory and disk space. WordStar, for instance, has its own paging routines to bring parts of documents in and out of memory. If the disks are of reasonable capacity, this offers no problem. However, for small systems with floppy disks and consequently small disk capacity, the amount of the disk needed for paging leaves little room for document storage. This causes, in some systems, the unfortunate situation in which a document that is being edited cannot be stored back on disk.

1.7.2 CPT 8000

CPT is a representative example of a commercial stand-alone word processing system. The Disktype 8000 has a page-size, monospace display, and two floppy disks to store files. CPT was the first word-processing system to offer an $8\frac{1}{2} \times 11$-inch white screen with black characters, simulating a piece of paper in a typewriter [SEYP79]. In fact, the typewriter metaphor is consistently applied. A few lines up from the bottom of the page is the *typing line*, meant to simulate the paper bail on the platen of a typewriter. Input takes place on the typing line only.

^A	Cursor word left
^B	Paragraph REFORM
^C	Scroll up screen
^D	Cursor character right
^E	Cursor up line
^F	Cursor word right
^G	Delete character right
^H	Cursor character left
^I	TAB advance
^J	Prefix HELP
^K	Prefix Editing
^L	FIND/REPLACE AGAIN
^M	RETURN
^N	Insert carriage return
^O	Prefix Formatting
^P	Enter control character
^Q	Prefix cursor editing
^R	Scroll down screen
^S	Cursor character left
^T	Delete word right
^U	INTERRUPT commands
^V	INSERT On/OFF
^W	Scroll down line
^X	Cursor down line
^Y	Delete line
^Z	Scroll up line
^JB	Defines REFORM
^JD	Print directives
^JF	Defines FLAG characters
^JH	Set HELP level
^JI	Command Index
^JM	Defines Tabs, Margins
^JP	Defines Place Markers
^JR	Defines Ruler Line

^JS	Defines Status Line
^JV	Defines text moving
^K0-9	Set/Hide Place Markers
^KB	Mark/Hide Block begin
^KC	Block COPY
^KD	Save file
^KE	RENAME file
^KF	Directory ON/OFF
^KH	Hide/Display Block
^KJ	Delete file
^KK	Mark Block end
^KL	Switch Logged drive
^KO	COPY file
^KP	PRINT
^KQ	Edit abandon
^KR	Read a file
^KS	Save file re-edit
^KV	Block move
^KW	Write Block to file
^KX	Save file Exit
^KY	Block Delete
^OC	Center cursor line
^OD	Display DOT commands
^OE	Soft hyphen entry
^OF	Set margins,tabs as exist
^OG	Paragraph tab
^OH	Hyphen Help ON/OFF
^OI	Set tab stop
^OJ	Justification ON/OFF
^OL	Set left margin
^ON	Clear tab stops
^OP	Display page break ON/OFF
^OR	Set right margin
^OS	Set line spacing

←	Deletes character left
ESC	Error release
RETURN	Hard carriage return
TAB	tab
NO FILE MENU-	(When no file is being created or edited)
D	Create or edit document file
E	Rename file
F	File Directory OFF/ON
H	Set Help level
L	Change logged drive
M	Merge-Print
N	Create, edit non-document file
O	Copy file
P	PRINT, stop print, start print
R	Run program
X	Exit to system
Y	Delete file

^OT	Display Ruler
^OV	Variable tabs ON/OFF
^OW	Word Wrap Off/ON
^OX	Release margins
^PM	Overprint next line
^PO	Enter non-break space
^Q0-9	Cursor to Marker
^QA	REPLACE
^QB	Cursor Block beginning
^QC	Cursor file end
^QD	Cursor right end of line
^QE	Cursor top screen

^QF	FIND
^QK	Cursor Block end
^QP	Cursor previous position
^QQ	REPEAT next command
^QR	Cursor file beginning
^QS	Cursor screen left
^QV	Cursor source (Block, Find)
^QW	Downward scroll
^QX	Cursor screen bottom
^QY	Delete to end of line
^QZ	Upward Scroll
^Q←	Deletes to front of line

Figure 15. WordStar commands. (From MICR81. Reprinted with permission.)

No standard cursor keys exist on the CPT. Rather, the space bar moves the cursor forward on the typing line and the backspace key moves it backward. There is no need for up and down cursor movement, since it is the document that travels up and down past the typing line. Therefore keys are provided to scroll the document up and down. Margins are set by moving right and left markers on the typing line. Five other keys specify character, word, line, paragraph, and page elements for commands like delete, skip, move, and insert.

CPT provides three input modes. Manual mode simulates a typewriter; when the user reaches the right margin, a bell rings. Wraparound mode provides automatic carriage returns when the right margin is exceeded. Hyphenation mode performs automatic hyphenation when a word reaches a system-defined "hot zone," using an algorithm aided by an exceptions dictionary.

One interesting feedback mechanism of CPT is its error message facility. In the center of the status line at the bottom of the screen is a 20-character area reserved for error messages. Rather than having terse error messages in this area, and as an alternative to removing some of the possibly offending text from the display to make room for a wordy error message, CPT rolls the lengthy error across this area like a captioned bulletin at the bottom of a television screen.

1.7.3 NBI System 3000

The NBI System 3000 is another popular commercial word-processing system. It has a stand-alone processor, with file storage on floppy disk. Its conceptual model is very similar to that of WordStar described earlier. The interface uses a combination of option sheets and function keys; the display is a mapping of a screen-sized viewing buffer to a full-screen viewport. The user alters the documents by driving a cursor around the screen with cursor keys, overstriking characters or using appropriate function keys to effect the changes. Like WordStar, the NBI System 3000 supplies the user with option sheets to show available operations. It does not, however, offer the help level commands of WordStar. In some cases, the user can operate without calling up an option sheet at all, and in other cases, as in WordStar, if the user is "faster" than the option sheet, it is not called up at all.

NBI presents an interesting alternative to the insert mode versus overstrike mode "controversy." When the cursor is positioned over a character, typing will overwrite that character. When the cursor is positioned over a space, typing will invoke insert mode and all characters to the right of the cursor will be pushed to the right as necessary. NBI provides an extensive search and replace facility, allowing the user to perform case-insensitive searches and replacements on a case-by-case basis.

Along with these advantages are several inconsistencies. The commands set is not consistent. The line delete command will delete the line in which the cursor is positioned, regardless of where the cursor lies in that line, while the word delete command will only delete a word if the cursor is positioned at the first character of that word. Although complete region selection and associated copying, moving, deletion, and storage are available, NBI provides no feedback as the areas are selected.

1.8 Integrated Environments

1.8.1 RIG, Apollo

The Rochester Intelligent Gateway (RIG) user interface [LANT79, LANT80] and the similar Apollo Aegis user interface [APOL82] are two examples of a relatively new trend in editing systems, one in which the editor is an integrated part of the interface presented to the user, rather than a user-invoked utility program.

Both the RIG and Apollo systems are based on the concept of a display or window manager as the primary interface to the system. These display managers give the user the ability to create windows on the display surface, move these windows around, and change their size. On the Apollo these windows can overlap; in RIG the windows do not overlap but simply partition the display screen (see Figure 16).

As in Star, the windows are meant to simulate pieces of paper on a desk. More

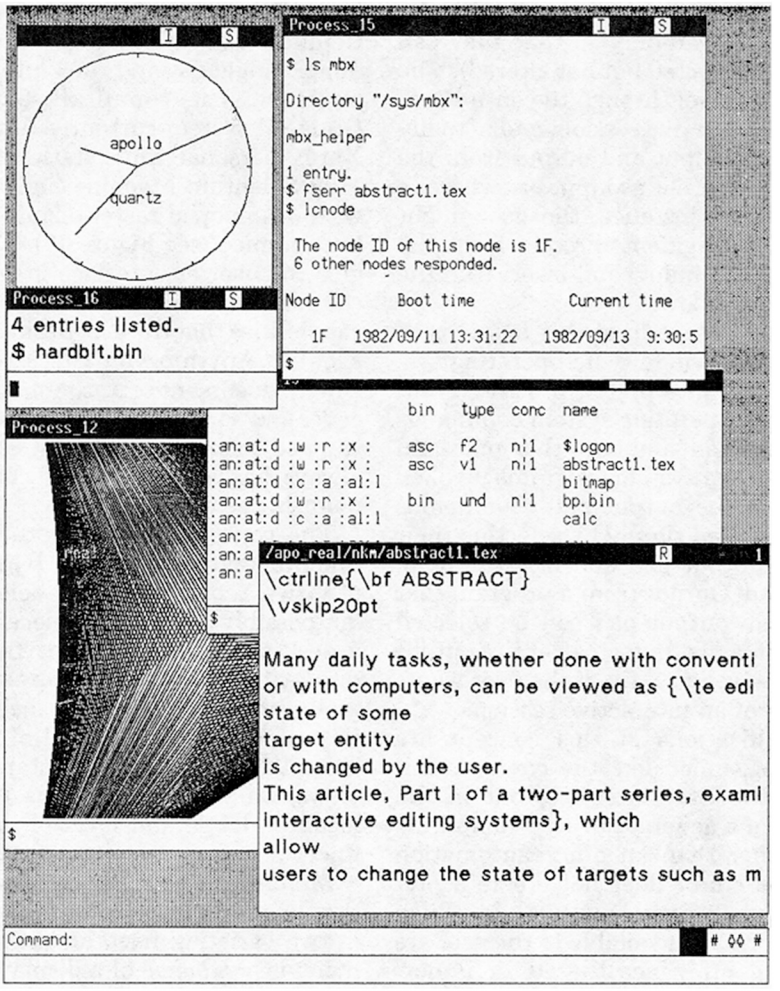


Figure 16. Apollo pads.

specifically, the window shows a rectangular portion of a two-dimensional *pad* (see Figure 16), an unbounded quarter-plane of text. Editing functions much like those in Irons model editors are supplied to manipulate items in the pad.

So far, the systems sound much like a standard multiwindow editor. In fact, one of the two types of windows, the *editing window*, fits that model exactly. The other type of window, the *process window*, is an editing window with an arbitrary user or system process attached to it. The process window has both a large *output pad* window on top and a smaller *input pad* window directly underneath. For interactive processes, input is typed into the input pad and, much like a typewriter, when a carriage return is pressed, this text scrolls up over the "typing bar" into the output pad. In addition, output from the interactive process is written directly to the output pad. The result is a complete transcript, simulating output on a hard-copy terminal, of the interactive session. Multiple windows on the screen may be active at once; the user may be reading electronic mail in one process window, editing the file referenced in the mail in an editing window, and compiling that file in another process window. Returning to our editing concerns, we note that the output pad allows all editing func-

tions (although often the system makes the pad's contents read-only so that they can be perused or selected but not altered). The user can now travel through the entire contents of an interactive session, and actually select previous input and output from the output pad and use it as input elsewhere in the system. For instance, the user might select a code fragment directly from the electronic mail window and insert it in the open editing window.

The major use made of this model is to tie to a process window the operating system command interpreter process. Now, the user types operating system commands into the input pad, and both this input and the output of the system programs invoked are stored in the output pad. Commands can be reexecuted simply by selecting their text in the output pad and inserting it in the input pad. Output from a program that is lying in an output pad can be selected and stored in a file. In fact, an entire output pad can be saved as a file if the user wants a transcript of an interactive session.

The ramifications of this concept are wide ranging, much like the concepts imparted by the Star interface, but in the framework of a general-purpose computing system, rather than an office automation system. The editor does not create a preemptive environment [SWIN74] in which functions normally available to the user are suddenly cut off. Since the editor is now above the command interpreter (rather than being an applications program invoked by the command interpreter), the user can freely issue system commands interspersed with editing commands. No longer does the user have to leave the editor to do something as simple as reading electronic mail or listing the files in a directory; the user simply switches windows momentarily, executes the appropriate command, and returns to the previous window.

1.8.2 Smalltalk-80

Smalltalk-80, a research product of Xerox PARC's Software Concepts Group (originally the Learning Research Group), provides an even more integrated environment than described above. In fact, the paradigm of overlapping windows was developed as part of the Smalltalk-80 project [LRG76]. Composed of an object-oriented programming language and an integrated user interface, the Smalltalk-80[6] [GOLA82, GOLA83] system currently runs on several Xerox personal work stations (the Xerox 1100 Scientific Machine and the Dorado) with bit-mapped raster displays and three-button mice (see Figure 17). The aim is to give the user an interface in which editing commands are always applicable and other capabilities that the user desires are at hand as well. Anything on the screen may be edited: document text, commands, program code, and so on. The user does not become trapped in modes (as in SOS above), but always has a full range of choices at any point in the editing session.

The conceptual model provided by the Smalltalk-80 user interface [GOLA82] is one of views, represented as labeled, rectangular, possibly overlapping pieces of paper on a desk top. A view is a particular way of displaying the information of a task or group of tasks for user inspection, alteration, storage, and retrieval of information. The views most used are standard system views, on which operations to alter size, location, label, and level of detail are defined.

Menus are the other important entity in the Smalltalk-80 user interface. They exist in two varieties: fixed and pop-up. A fixed menu is a *subview* of a displayed view; the user moves a cursor over the menu items with a mouse and selects an item by pressing the leftmost mouse button. This action highlights the selection. Releasing the button invokes the selected command. A pop-up menu appears directly under the cursor when the user holds down one of the other mouse buttons. As the cursor is moved around the menu, the item underneath the cursor is suitably highlighted. Upon releasing the button, the item is selected and the menu disappears.

Since the screen space available to present a view may not be large enough to contain all the information that needs to be presented, Smalltalk-80 provides the *scroll bar*, a special type of menu that allows the

[6] Smalltalk-80 is a trademark of the Xerox Corporation.

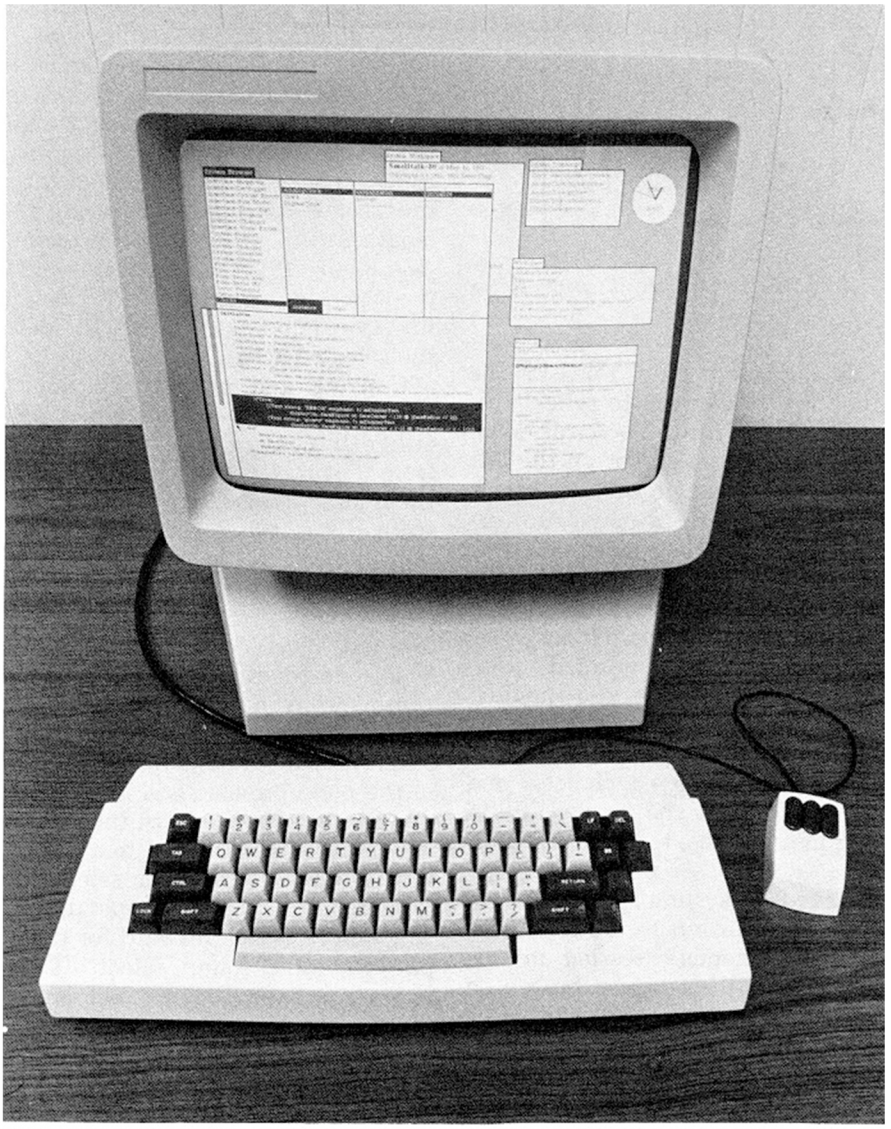


Figure 17. Smalltalk-80 screen and mouse. (Courtesy Xerox Corporation.)

user to select what portion of the view is to be made visible. It supports three general operations: scroll up, scroll down, and jump. The scroll bar is displayed as a vertical white rectangle that appears out to the left of a view's rectangle when that view is being used. This white rectangle represents the document continuum. Inside this rectangle is a smaller, gray rectangle. This represents the viewing buffer of the document, the amount of the document that is currently being viewed. For example, in Figure 18a, the gray bar indicates that about the last half is being viewed; in Figure 18b, the first half is being viewed; in Figure 18c, the entire document is being viewed. To jump to a particular place in the document, the user puts the cursor in the gray rectangle, holds down the leftmost button, and drags the gray rectangle up and down by moving the mouse. Moving the rectangle simulates changing the placement of the viewing

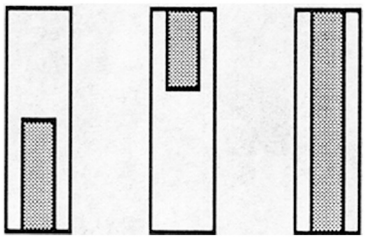


Figure 18. Smalltalk-80 scroll bar. (Courtesy Xerox Corporation.)

buffer on the document. Upon releasing the button, the view is changed to reflect this viewing buffer placement. To scroll down (up) the user places the cursor on the left (right) of the gray rectangle, and the cursor automatically changes to a down (up) arrow. Upon pressing the leftmost button for downward scrolling, the line of text closest to the line of text at the top of the view is moved to the cursor position. Upon pressing the leftmost button for upward scrolling, the line of text closest to the cursor is moved to the top of the view. Other menus include *confirmers*, which allow a user to select one of two displayed choices, and *prompters*, which allow the user to "fill in the blank" in response to a question or message.

To edit text in a system-specified view, the user makes extensive use of the mouse and the supplied menus. Pressing and releasing ("clicking") the leftmost button on the mouse causes a caret to appear in the intercharacter gap closest to the cursor. This essentially selects a zero-length string. If the user continues to hold down the leftmost button, all the characters between the initial caret and the current position of the cursor are highlighted in reverse video. Releasing this button causes the highlighted text to become the active selection. Two clicks on the leftmost button while the cursor remains stationary select the word on which the cursor lies, unless the cursor is at the beginning or end of the document, in which case the entire document is selected, or unless the cursor lies just after (before) a left (right) parenthesis, square bracket, angle bracket, single quote, or double quote, in which case the text between the delimiter pair is selected.

Replacement fits naturally with the concept of selection. Inserting characters is simply a form of replacement. The user moves the cursor to the gap in which the string is to be inserted, does a single click to get the caret, and begins typing, which causes characters to be inserted at this spot: the user is essentially replacing the zero-length string with the string being typed (see Figure 19). If a larger area is selected, the typed-in text replaces the selection (the first typed character causes the deletion of the selected area).

Copying and moving are done analogously. The proper selection is made and either the copy or the cut button is selected from the pop-up view menu controlled with the middle mouse button. Both commands store the selected text in a paste buffer; copy leaves the selected text unscathed, while cut deletes it. Now the user is free to perform any action; the copy and cut are finished. When desired, the user simply selects a destination point, selects the paste command in the menu, and the proper insertion is made. If the user has just selected and replaced text, again, another command in the pop-up menu, will hunt for the next occurrence of the pattern that was selected and replace it with the text used for the replacement. If the user selects text and then issues the again command, the system will simply search forward for the selected pattern, and if found, select it so the user may cut it or replace it (see Figure 20). An undo command allows the reversal of the previously issued command.

Most Smalltalk-80 commands are issued by either clicking a mouse button, or by the dual motion of first pressing a mouse button to select a menu item and then releasing the button to invoke the associated command. When a command is not available this way, the user simply finds some empty view space, types in the appropriate command, selects it, and then issues the doIt command. Later on, one may come back, edit that command in the normal way, and reissue it. Thus commands are edited as easily as text; indeed, commands are simply text. The printIt command is identical to doIt, with the exception that any output result is placed immediately after the com-

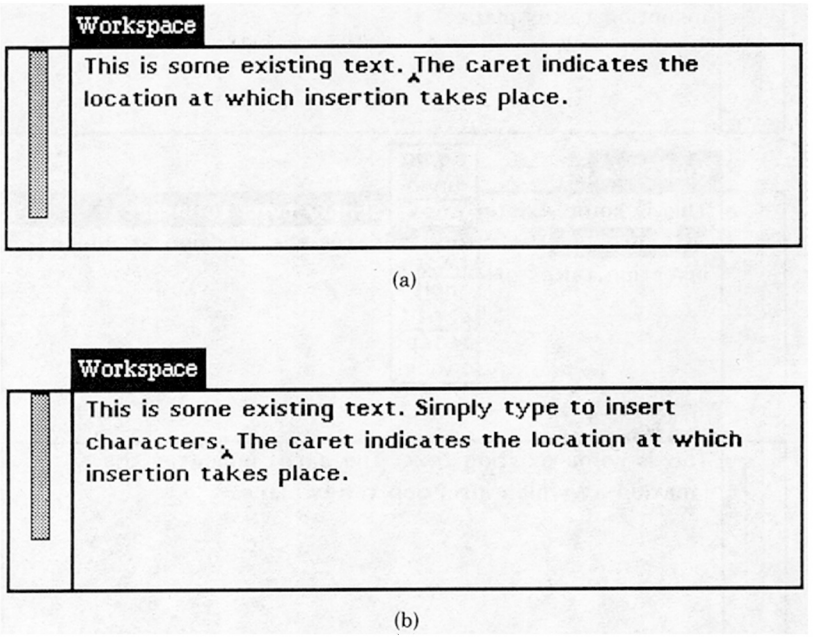

Workspace
This is some existing text. The caret indicates the location at which insertion takes place.
(a)
Workspace
This is some existing text. Simply type to insert characters. The caret indicates the location at which insertion takes place.
(b)

Figure 19. Insertion in Smalltalk-80. (Courtesy Xerox Corporation.)

mand text to which printIt was applied and this result is automatically selected.

The environment supports a wide range of tools for the development of Smalltalk-80 programs. While most are beyond the scope of this paper, we mention browsers as an interesting method for traversing hierarchies. Objects in the Smalltalk-80 language are built hierarchically. For instance, number is a **class name** that is in the browser **class category** numeric objects. Similarly, the operation of absolute value is a **message selector** that is in the browser numeric-number **message category** arithmetic. To find information about these quickly, the user simply selects numeric objects in the class category subview in the browser, which brings up the appropriate classes in the class name subview. The user next selects Number in the class name view, which brings up the appropriate method categories in the message category subview. The user than selects arithmetic, which brings up all appropriate messages in the message selector subview. Choosing one of these message selectors, for example abs, causes the Smalltalk-80 procedure for that method to be brought up in the editing view below (see Figure 21a). The user is free to edit this procedure using the editing facilities described earlier. Pointing to an item in a subview further up the hierarchy replaces the subviews below, and allows the user to browse through new definitions. For instance, as in Figure 21b, selecting truncation and round off in the class category subview causes a new message category subview to be generated and the message selector subview to be changed. Subsequently, selecting truncated causes a new procedure to appear in the editing subview.

The browser is important not only for its convenient techniques for tree traversal, but for its notion of letting a user browse through an entire collection of information, examining and editing it at will. A browser-like interface would be attractive in other environments as well, such as that of examining a file system hierarchy or traversing a hierarchically structured menu system. Indeed, the access to local and remote files is provided in the Smalltalk-80 system through appropriate browsers in which file name patterns are specified and file names are selected for reading, retrieving, or editing.

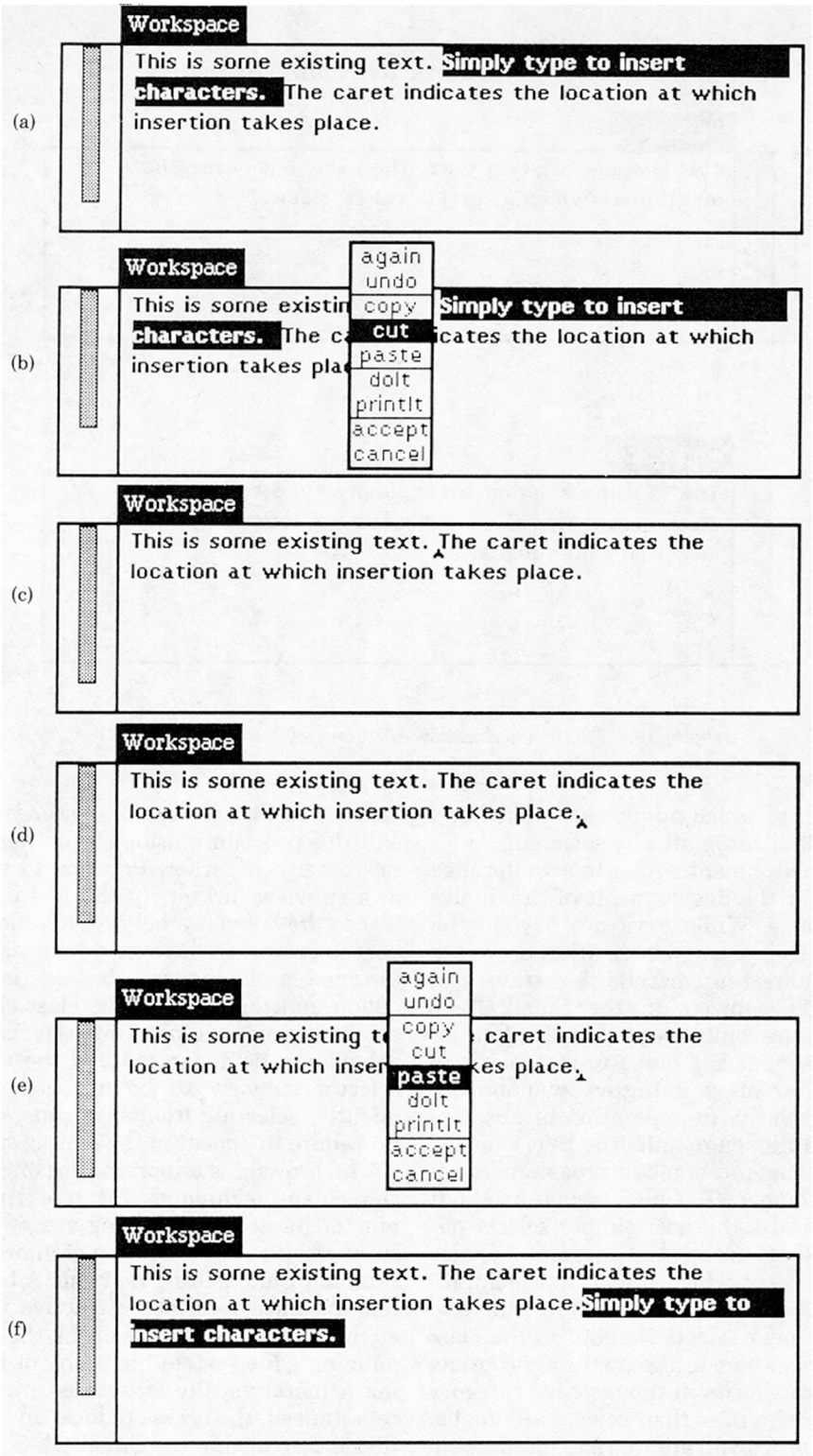
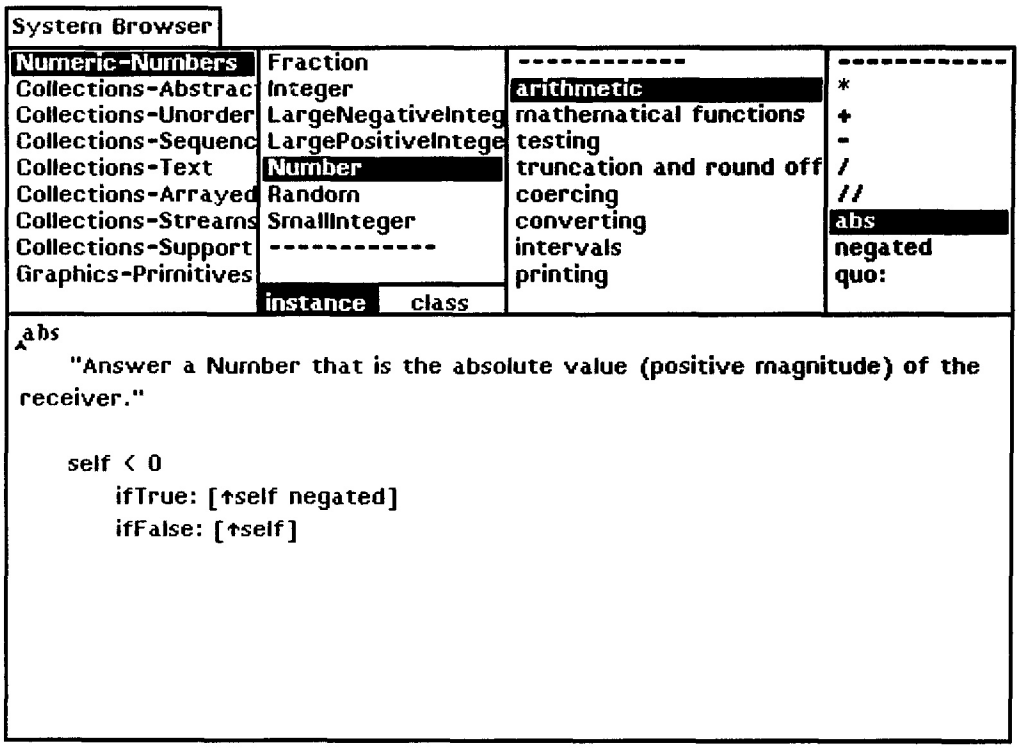
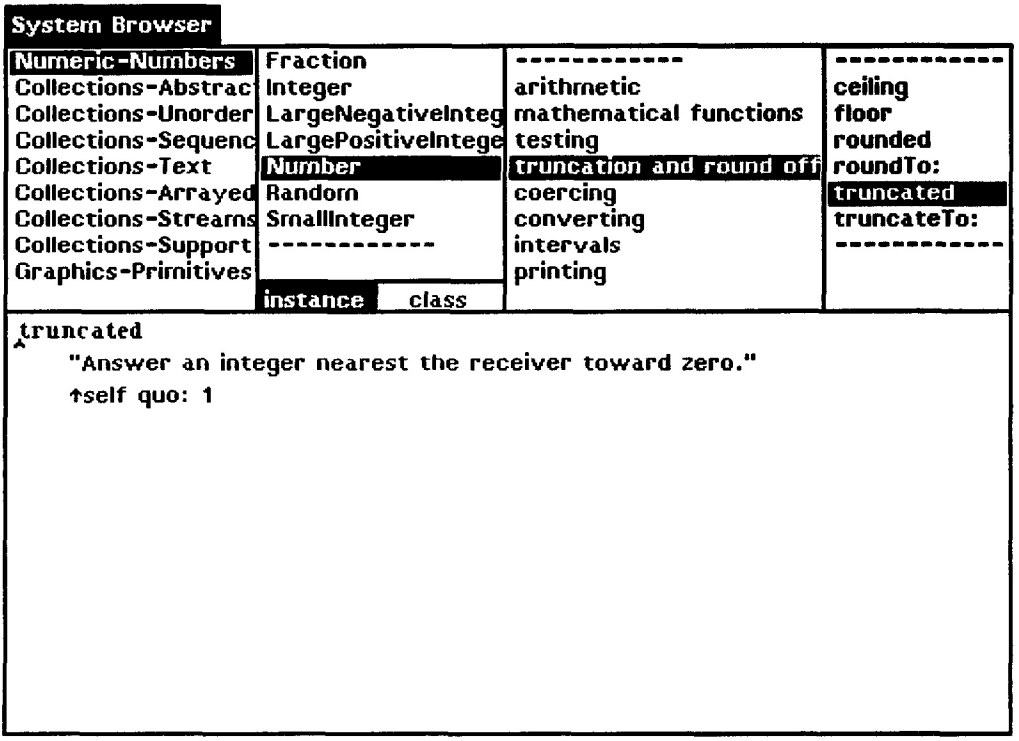


Figure 20. Cut and paste in Smalltalk-80. (a) The user selects the text to be moved (b) The user holds down the middle mouse button and the pop-up menu appears. The user moves the cursor over the cut button, which is appropriately highlighted. (c) Releasing the mouse button causes the selected text to be removed. (d) The user selects the reinsertion point with the left mouse button. (e) The user holds down the middle mouse button and the pop-up menu appears. The user moves the cursor over the paste button, which is appropriately highlighted. (f) Releasing the middle mouse button causes the previously cut text to be pasted in at the selection point. (Courtesy Xerox Corporation.)



(a)



(b)

Figure 21. Smalltalk-80 browser. (Courtesy Xerox Corporation.)

2. ISSUES

2.1 The State of Editor Design

While much is written about "the desirable human interface," most of it (unsupported) personal opinion, very little experimentation in determining the optimal editor interface has been done. Typically, editor design is based not on concrete experimental results, but on market pressure to design systems that conform to today's often worn technology (such as 24 × 80-character terminals, and half-duplex communication to time-sharing systems). Rather than concentrating on desired functionality and ease of use, the editor designer is then forced to devote large amounts of time to molding the user interface to the constraints of particular classes of limited input and output devices, producing a far from optimal interface.

In our editing model, the lexical phase of the command language processor, which composes tokens from lexemes, is followed by a syntactic phase, which parses sentences of these atomic tokens. In principle, we want each token's appearance and meaning to be unambiguous in all contexts, and its user image to be unique, easily remembered, and unobtrusive. For typed command languages, this is not the case: the user must correctly spell or abbreviate the needed tokens, usually from memory, and the system must be in the appropriate "command" mode so that command tokens are accepted as such, and not as literal text. In control-key interfaces, tokens are composed by overloading the alphanumeric keyboard with control or prefix keys to form cryptic combinations.

If tokens are atomic entities unto themselves, why do we need a lexical component to compose them at all? In fact, the token composition phase is one of the most treacherous parts of a user interface, and is, for all practical purposes, unnecessary with modern technologies. The pure function key interface, for example, assigns a token to a particular key; composition of tokens is not necessary. The flaw in this technique is that the number of function keys grows linearly with the number of tokens; a many-function editor would need a massive, untenable keyboard. One type of interface that begins to satisfy the criteria of atomic tokens without incurring the expense of linear growth of input devices is the menu interface, more explicitly, one that uses pop-up menus in temporary viewports and selection devices like the mouse (see Smalltalk-80 in Section 1). In general, menus supply composed tokens in a form the user easily recognizes. No memorization of tokens is needed, since the necessary images appear as needed; the user simply points to the appropriate image and selects it. The menu presents to the user only those tokens that are viable at any particular time, unlike the function keyboard, which is obliged to have all tokens at all times to avoid overloading. The menus are unobtrusive; when not in use they disappear and only pop up when called. Another interface satisfying this criterion is the Star interface. Star eliminates the problem of overloading commands by having a small set of general-purpose function keys such as open, move, and find that provide the majority of tokens needed, a rich set of icons that serve as tokens as well, property sheets and option sheets that provide consistent access to tokens as well as a consistent method for composing new tokens in these sheets, and finally, window-specific menus containing selectable command tokens.

Design techniques pose another problem. Many editors in production today have been designed **by** programmers **for** programmers, and have been foisted upon the general public with little apparent regard for its needs. Many others appear to have been designed by nonprogrammers for nonprogrammers, and show little evidence of proper software engineering and language design principles, such as consistent user instruction sets and consistent syntax. Clearly few editors have been designed by rigorous examination of reasonable choices in interface and functionality, and even fewer are backed by a well-explained conceptual model. Rather, editing design has been ad hoc, with the editor often becoming a potpourri of contradictory techniques and functions, copying and inheriting poor design from previous systems ("we can always change it or write another one"). It is time that editor designers, like programming language designers, commit their conceptual models and user interfaces to paper before implementation. This requires extensive

search of the literature, analysis of alternatives, and experimental validation of ideas, all traditional actions in science and engineering but disappointingly rare in this field.

2.2 The Modeless Environment

There has been much recent interest in so-called "modeless environments" [TESL81]. In fact, the term "modeless" is a bit of a misnomer since no system can be truly without modes. What is intended is that modes be minimized, and that designers move away from implementing special-purpose context-sensitive states and commands of the type that SOS has. The primary problem with modes is that they lock the user into a specialized and typically highly restricted functionality while in the mode, preempting the use of the normal set of functions and thereby severely limiting flexibility. With current techniques for command specification there is often a second problem, that of assigning different meanings to user actions as a function of the mode, as is the case in overloaded keyboards.

The goal of the modeless editor is to allow the user to have the flexibility to travel and to select operands without having to commit to a particular function and the particular options that it allows. In particular, the postfix form of command specification in which the operands precede the command is more conducive to minimizing modes than the prefix form. The latter is typically used to put the user in a temporary mode and then prompt for the operand(s): for example, move puts the user in a mode that requires the user to specify the source and then the destination. This style of guided dialogue, while useful for novices, is often frustrating and annoyingly time consuming for experts. Furthermore, it enforces sequential specification of multiple operands, without providing the ability to edit them. Worst, of course, is that the user is locked into the dialogue, and cannot leave to browse or to collect information with which to complete the command (see Tesler's painfully amusing example in TESL81).

With postfix syntax, the user spends most time browsing and selecting without committing to a particular function. When the function is specified, it is executed indivisibly and the user is back immediately in the familiar and universal operand selection "mode," free to browse, to create other window/viewport combinations, and to select and revise selection of operands before specifying another operation. It would be possible to allow the user to escape from a temporary operation mode in prefix interaction, but a great deal more status saving would be required.

In the case in which certain commands require more than a single operand followed by an operator, there are several alternatives in a postfix system: (1) split a compound operation into smaller primitive operations that fit the postfix constraint (as in two single-operand cut and paste commands to replace move); (2) allow multiple selection of operands, although this becomes difficult if the order of specification is important; (3) allow the use of familiar editing functions for specifying parameters and for setting attributes by letting the user fill out a form and then execute a command based on this form; and (4) temporarily switch to infix specification. For an example of the last alternative, a move command that takes both a source and a destination would be specified in the normal way by selecting first the source and then move. Then, the system would enter temporary move mode, ask the user to select the destination, execute the move command, and return to the familiar operand selection mode.

The Star system uses the third alternative above, providing option sheet forms that do not preempt the user (as shown in Section 1). To issue the find command, for example, the user presses the find key, gets the option sheet, fills in the appropriate parameters, and then issues a command that actually does the search. Using the form metaphor, the user has the ability to select information from other parts of a screen as input to the form, and of course, has the ability to edit the form as well. In fact, the form can be used to simulate a dialogue. As the user fills in particular information or toggles particular attributes, the system can provide further fields to be filled in. If the user goes back and edits one of the fields, all of the field values that

depended upon one particular field value may be undone. (See the example of Star query-replace in Section 1.) Not only does the form metaphor simulate interactive dialogues, but it obviates the sequentiality and noneditability problems of conventional dialogues. Why then is filling in a form not considered a "dangerous mode"? In fact, form filling is a mode of sorts, yet familiar functions can be used to edit it. Most importantly, the user can leave the context of the form, issue other commands, and return without loss of context.

2.3 Instant Editor/Formatters versus Batch Formatters

The classical separation between form and content enforced by batch formatting is becoming increasingly less desirable. Space and layout constraints often force alteration of content to make text fit. Furthermore, text can be interpreted in a surprisingly different way when typeset than when it is printed as draft copy on a line printer. This, of course, is the reason it is typeset at all. That computer scientists, reporters, copy editors, and even professional printers have tolerated the system of marked up alterations and specifications on typewritten copy or bad facsimiles of a final typeset galley is a result of economics and not an implicit confirmation of that system. The typesetting conventions that make it easier to understand text in printed form make it correspondingly easier to understand the on-line form. For all these purposes and especially for complex formatting tasks (tables, equations), interactive formatting is clearly highly desirable.

Yet, there is a strong camp advocating continued use of batch formatting systems, with possible soft-copy review, to allow maximum flexibility and power, especially in terms of multiple interpretations of markup tags (e.g., those that indicate different document styles and output devices). Allen et al. [ALLE81] advocate the use of soft-copy output that is later more precisely formatted by a document compiler. They contend that the interactive user does not need a finely formatted document, but simply one that approximates the final printed result. The interactive system does not need to perform expert formatting; this is left to a batch document compiler. The underlying notion is that no matter how accurate interactive formatting systems can become, those (batch) methods that spend more time will produce higher quality output.

Still, interactive editor/formatters seem to have compelling advantages over editors that have a separate, editable representation for formatting effects, and certainly over the separate interactive editing/batch-compiling method. The ability to experiment with different formats is clearly invaluable to both author and transcriber, providing that there are no serious restrictions resulting from this facility. Having to "program" formatting effects is a mental burden and requires sophisticated, complicated code all too often; debugging a sequence of formatting codes is difficult unless a formatted copy of the same document exists for comparison.

On the other hand, the problem with some interactive editor/formatters, often called "what-you-see-is-what-you-get" editors is that, as Brian Kernighan has remarked, "what you see is all you've got." That is, it is just as uninformative and unhelpful to give a user a view of a beautifully formatted document with no clues as to how or why the formatting was effected as it is to give the user a file laden with complicated formatting codes without the rules for what these formatting codes will do. However, the stripping of formatting information is not necessary to interactively produce a finely formatted document. Referring back to our editor model in Part I, Section 1, we can interpret the finished document page as simply one of many useful views, but not the only one to which the user should be restricted. In fact, the property sheet of Star and the margin tags of ETUDE are simply specially tailored views of the document data structure. In particular, structure editing can be nicely done on representations that stress the structure and suppress formatting information—one can rearrange sections in an outline much more easily if only the section headings and the first line of each section are displayed, as in NLS. Also, as Jan Walker has pointed out [WALK81a], it is

often important to know why a particular formatting effect is apparent; it is useful to be able to interrogate and alter the higher level document object specification that caused the effect.

The principle of multiple views, one that has been sorely underutilized in the hundreds of editors that have been created, shows that a completely reasonable solution to satisfy both camps is to provide whatever views each desires. The batch community might get a view that allows them to edit textual descriptions of formatting, equations, and tables, while the interactive community might be given a view that allows interactive specification of tables and equations, as well as of the traditional simple (and local) formatting effects. Except for the additional implementation time, there is no reason to restrict the user to editing a single interpretation or view; the multiple-viewing principle needs to be adopted in systems of the future.

2.4 Structure/Syntax-Directed Editors versus "Normal" Editors

With the increase in the number of structure editors, several designers have explained the rationale behind what seems at first to be a restrictive concept.

Advocates of structure editors claim that the specification of target data as well-connected, well-defined units enhances the user's powers of creativity and composition. Engelbart, describing a key idea in NLS, writes:

> With the view that the symbols one works with are supposed to represent a mapping of one's associated concepts, and further that one's concepts exist in a "network" of relationships as opposed to the essentially linear form of actual printed records, it was decided that the concept-manipulation aids derivable from real-time computer support could be appreciably enhanced by structuring conventions that would make explicit (for both the user and the computer) the various types of network relationships among concepts . We have found that in both offline and online computer aids, the conception, stipulation, and execution of significant manipulations are made much easier by the structuring conventions ... We have found it to be fairly universal that after an initial period of negative reaction in reading explicitly structured material, one comes to prefer it to material printed in the normal form. [ENGE68, pp. 398-399. Reprinted by permission AFIPS Press]

Burkhart and Nievergelt [BURK80] concur with the view that while the structuring seems to be a restriction on the user (especially the novice), who may not want to be forced to keep track of the data hierarchically, the structuring would ultimately be performed anyway, "into chapters and paragraphs, procedures and modules, subpictures and patterns, as the semantics of the data may suggest."

Using his Cornell Program Synthesizer as an example, Teitelbaum claims the value of the syntax-directed editor to be the fact that a program being developed is always structurally sound, even if not complete. The use of structural templates eliminates mundane program development tasks. Indentation and prettyprinting are automatic, typographical errors are possible only in user-typed phrases, not in system-supplied templates, and such errors can be easily caught at run time. The templates save keystrokes, as one typed command may generate a long template. Place holders in the templates act as prompts, guiding the user along the proper path. The user never needs to get mired in low-level syntactic detail—the constructs are always conceptualized as abstract units, not as streams of tokens.

On the other side, Woods [WOOD81] claims that a good "standard" editor can do 95 percent of the program editing that a syntax-directed editor can, at much smaller development and computation costs. He claims that syntax-directed editing constrains the user interface, complicating operations that are normally easy in a standard editor. This is true of some operations in the available interfaces today, but is not an intrinsic restriction on future interfaces. He further claims that the syntax-directed approach promotes a multitude of editors. This is only partially true; editor generators such as the Cornell Synthesizer Generator, the GANDALF/ALOE project [NOTK79, FEIL80, MEDI81], and *sds* show that editors for very different targets can have the same basic editing operations. In fact, the regularity of the structure editor introduces the ability to produce formal descriptions to generate special-purpose editors. This par-

allels the trend to create parser generators and compiler generators from formal descriptions. Woods's claim that the representation and editing of a program as a parse tree makes an editor harder to implement is certainly true; yet, again, the syntax-directed editor eliminates the need for a parser completely.

Notkin points out that syntax-directed editing may change the way that programming is taught and described [NOTK79]. For instance, "nitty-gritty" details such as placement of statement delimiters like semicolons could be eliminated entirely, since the templates will carry whatever information is necessary to create a structurally correct program.

Morris and Schwartz [MORR81, p. 29], among others, contend that syntax-directed editors are "profligate consumers of computer resources .... Parsing consumes processing power, the parse tree devours storage, and there is no solution but to supply plenty of each." With high-performance personal work stations that possess virtual memory, however, this is no longer a very important consideration.

We believe that given adequate machine resources and a well-engineered human interface (perhaps a two-dimensional syntax with interesting icons), there are many reasons to prefer a syntax-directed approach to editing. However, we know of no formal determination of trade-offs between "normal" editors and structure/syntax-directed editors. We hope that controlled experiments in this area will result in the necessary data for an objective, informative evaluation of the utility of structure/syntax-directed editors.

3. CONCLUSION

3.1 Desiderata for Today's Editor

As in programming languages and most computer systems, the "desirability" of the syntax and semantics associated with an interactive editor is largely a matter of individual taste. Often, however, constraints imposed by old techniques and methodologies and acceptance of outmoded technologies (e.g., 300-baud "glass teletypes") force inferior modes of communication. Without being unduly constrained by the limitations of old-fashioned technology, we suggest our design criteria for an ideal interactive editor within the limits of today's modern technology. Such an ideal editor should have

- A well-defined, consistent conceptual model, rather than a seemingly haphazard organization. The user must be familiar and comfortable with the "philosophy" behind the system.
- Documentation, both on-line in a help facility and off-line in manuals, which explains the conceptual model as well as the details of the user interface and the functions of the system.
- A clear and concise user interface that is easy to learn and to use and that provides consistency across different targets such as text, pictures, and voice. Indeed, a good test of an efficient and pleasing interface is that authors will use the system to compose and revise manuscripts themselves. We believe that the author should not need to involve others (experts or "wizards" to advise, secretaries to make changes) in any phases of document creation or editing.
- An "infinite" undo/redo capability, enabling an author to experiment without concern of loss or damage to a document.
- Fast response time. No noticeable delay should exist for all but the most complex commands.
- Powerful facilities, with few restrictions and exceptions, to make possible everything that one can do to hard copy with red pencil, ruler, scissors, and tape.
- Facilities that take advantage of computer capabilities to compensate for human limitations. Examples of existing facilities include global substitution of one pattern for another throughout a document, replication of a standard phrase or paragraph, and automatic renumbering of sections or references after (or while) a file is edited. Some of these facilities may duplicate human processes; others may be functions that are available only with the power of a computer.
- User access to shared information and files under controlled conditions (useful for a pool of researchers or documentors working in the same area, or for common access to dynamically updated management information).

- The ability to mix targets, such as text, graphics, programs, and forms, with ease.
- The ability to have multiple contexts on the same display surface, allowing the user to browse through and use a large assortment of familiar utilities and documents in an editing session. The editor should not force the user into a smaller, less powerful environment in which normally provided system functions are preempted. In fact, the editor should be part of a larger, integrated environment, allowing the user, in the middle of an editing session, to obtain information by looking through the file system, to use a desk calculator utility, or to retrieve an electronic mail message or a piece of data from a database system, with transparent return to the editing context.
- The ability to edit a close facsimile of the final composition, layout, and typography of the document without significant impact on computer response time.

3.2 Standardization

Several ANSI (American National Standards Institute) and ISO (International Organization for Standardization) committees (ANSI X3J6 and X3V1, ISO/TC 97/SC 5/EGCLPT) are considering standardization of generic markup languages, text-processing languages, text formatting, text editing, and text structures. We consider the agreement upon a standard editor, unfortunately, as unrealistic at this time. What is needed first is a standard reference model for editing (e.g., one based on the framework in Part I, Section 1). The acceptance of at least an interim reference model on which to base further research and development of editors will be helpful for a productive interchange of ideas in the editing field.

Another step toward standardization would be the definition of a set of operations, called a virtual editing protocol (VEP) [MAXE81, WILD82] that acts upon any medium, such as text, graphics, or voice. The VEP would not define **how** operations are performed on the medium, but simply what generic operations can be done on all media. The virtual editor for each medium accepts the medium-independent VEP and translates it to medium-dependent operations.

3.3 The On-Line Community

Just over ten years ago, in our first survey of the field, we concluded

> On-line composition and editing of programs, coupled with interactive debugging, has become an established, cost-effective use of computers. Similarly, minor text editing, such as the correction of typographical errors in memoranda, is cost-effective, since only teletypewriter consoles and minimal service from the CPU are required. In contrast, the imaginative use of computers for on-line composition and extensive manipulation of free-form text is still in the early stages of experimentation and user conversion. This is due partially to the high cost of CRT terminals that provide the human factors essential to general-purpose editing, partially to the high cost of system resources and implementation time for the sophisticated programs required, and partially to the long time required to wean users from traditional off-line hard-copy processes.
>
> Hardware prices are coming down steadily, however, and as more users switch to on-line thinking, creating, and manipulating, the use of computers will become increasingly accepted (and judged cost-effective), hopefully to the same extent that numerical and data processing applications are already considered to be a legitimate use of the computer. [VAND71b, p. 113]

In the ensuing decade, this hope for the large-scale introduction of text processing was realized, as a result of rapidly decreasing hardware prices, the acceptance of word-processing systems in business, the introduction of personal computers in hundreds of thousands of homes and offices, and the rapid growth of software for interactive editing. Regrettably, much text processing today is still done by transcription from hard copy rather than by authors composing and editing on line. This is due in part to generally poor interfaces (limited windows on alphanumeric screens and transcription-oriented software), and in part to cultural resistance: typing, unfortunately, is not a universal skill and is still too often considered an inappropriate activity for executives and other high-level decision makers.

For the remainder of the decade and indeed the century we see an ever-increasing infiltration of editors. In one form or another they will become a fundamental

tool of modern communication in all walks of life. They will be the key user interface to the work stations that will be used in the office, the classroom, the home, and any other place in which information is entered, edited, and communicated. They will increasingly be used for do-it-yourself, high-quality document production with sophisticated typesetting effects, as well as for on-line browsing, studying, composing, and communicating.

As terminals and work stations become widely available and personal (i.e, become part of one's office or study furniture that, like a clock radio, are rarely, if ever, turned off) increasingly more of one's daily activities will be accomplished via the computer, increasingly less via traditional paper and mechanical communication. There is, in fact, a kind of "critical mass" phenomenon, in which a knowledge worker switches from hard copy to soft copy for most purposes, given the full-time availability of powerful local services and a high-performance information- and resource-sharing network. At that point, the Bush–Engelbart–Nelson visions of on-line communities will become commonplace realities and editors will be the key interface to all manner of document preparation and communication. We expect these editors to be suitable both for stream and structure editing, and for targets as diverse as text, pictures, and voice. The more intelligence the editor has of both the form and content of the manuscripts, the more powerful its capabilities will be.

Just as advances in technology in the past decade have provoked a marked change in editors, so will advances in inter-computer communication, speech synthesis and understanding, and character and handwriting recognition again change the way in which editors are implemented.

Imagine the following scenario. Families, businesses, and individuals will receive a symbolic computer address much as one receives a telephone number. A user anywhere in the world with access to this address will be able to access the files at that network address as if they were on that user's own machine (subject to any confidentiality and security restrictions imposed). Interdocument links will be made easily by including this user address as the first search criterion in the link address. Multiperson collaborations will become economically and technically feasible, and will make distributed knowledge work an attractive alternative to physical travel. Tymshare's AUGMENT and Nelson's Xanadu[7] system are ongoing projects to bring these concepts and many other adventurous document organization ideas to the general public. Nelson provides both an interesting history of the development of hypertext systems and a description of the Xanadu technical and organizational plans in NELS81.

Each user will have a personal work station with a high-resolution (several-hundred-points-per-linear-inch) bit-map display packaged in a flat, notebook-style package easily moved about a desk or carried in a briefcase [LRG76, KAY77, GRID82]. Interaction may be done in many ways. A wireless mouse or a touch-sensitive screen will allow for cursor movement. The cursor may be used not only for selection of entities from a menu, but also for drawing proofreader's symbols on a document or for entering text into the document. A symbol recognition program will understand the drawn symbols and perform the appropriate operation. If a symbol is inscrutable or ambiguous, the editor may notify the user using voice output. The user will have the choice of redrawing the symbol, or alternately, vocally inputting the commands (as well as, later, even natural-language text). While currently experimental and unportable, the use of eye-tracking schemes [BOLT80, BOLT81] may allow the editor to determine at what (large) area the user is looking, enabling it to correctly understand commands such as "delete **this** paragraph."

Before the turn of the century, the editing systems are likely to have taken the place of pen, paper, and typewriter—and not only for manuscript composition. For example, banks will have editors with preprinted "forms" that the user fills out using a keyboard or even natural handwriting. Documents will be interactive, compiled on demand especially for the requester. They will be further personalized with on-line

[7] Xanadu is a registered trademark.

annotation. Most important, schoolchildren will learn to both read and write with the editor and a nationwide library of on-line books.

In fact, much of the technology for the near-term editor of the 1980s is in place. Among the hardware are bit-mapped personal work stations, pointing devices, precision laser printers, and digital phototypesetters; among the software are multiwindow text and graphics editors, interactive formatters/typesetters, iconographic communications, and modeless environments. The key issue of the 1980s is the willingness of users and manufacturers to discard existing techniques for even better-researched, better-understood, and better-developed metaphors of user interaction.

In the broadest sense, most actions that people perform are editing operations of one form or another. In moving a car from here to there, making a shopping list, or playing chess, a person modifies or **edits** the state of some entity. In computing, most of the actions that people perform are editing operations as well. It is inevitable that the interactive editor will soon enter a new generation, a generation in which it forms the primary interface to the computer.

POSTSCRIPT

The majority of this document was edited using the *bb* editor running on a VAX 11/780 under Berkeley UNIX 4.1. Some parts of the document were occasionally edited using the Apollo editor, BBN's PEN editor, and Brown's CMS Editor. Besides these, the authors at one time or another have used *ed*, ex, vi, EMACS, SOS, the Cornell Program Synthesizer, FRESS, NLS, TECO, XEDIT, WordStar, Bravo, and Star.

Formatting was initially done using the TROFF package under UNIX. A revised version was translated to TEX by the use of keystroke macros in *bb* and through hand translation for some parts. No interactive formatting was available; the authors had to rely on hard-copy printouts from a Varian electrostatic printer-plotter (approximately 5 pages/minute) to see the formatting that the TEX codes had produced. The text consists of approximately 100, single-spaced, 12-inch high pages. With around 50 drafts of the paper over more than two years, we have regrettably used just under one mile of paper to produce a final draft, excluding the reams of paper used in duplicating review copies.

Communication of machine-readable files and electronic mail was done through the *uucp* inter-UNIX telephone network and through the Arpanet.

ACKNOWLEDGMENTS

The authors would like to thank the numerous people who have contributed their time and thoughts to the production of this paper, especially Steve Derose, Doug Engelbart, Joan Haber, Phil Hutto, Barb Meier, Steve Reiss, Terry Roberts, Katy Roth, David Smith, Tim Teitelbaum, Jan Walker, and Nicole Yankelovich. We also thank the numerous researchers whose systems we have used as examples, and apologize for any misrepresentations that may appear. A special mention goes to Trina Avery and Debbie van Dam for their outstanding copy editing and proofreading of multiple versions of the document, and to Steve Feiner, for his customary incisive comments and technical questions that resulted in more accurate explanations and descriptions. Janet Incerpi cheerfully answered our questions about and suffered our tirades over TEX. Rachel Rutherford's meticulous reading pointed out hundreds of places where we could clarify points for the reader; her delightful comments on the final draft were an important barometer as to how well we were achieving our goals. We gratefully acknowledge the reviewers, whose detailed critiques guided us toward a completely reorganized, clearer document. Finally, many thanks go to Adele Goldberg, who provided us with patience and understanding beyond the call of duty, immediate turnaround despite our somewhat more sluggish speed, and most important, numerous critical and editorial comments that improved the paper.

This research was sponsored in part by IBM under a research contract.

REFERENCES

ACHU81 ACHUGBUE, J. O. "On the line breaking problem in text formatting," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 117–122.

ALLE81 ALLEN, T., NIX, R., AND PERLIS, A. "PEN: A hierarchical document editor," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 74–81.

APOL82 APOLLO COMPUTER INC. *Apollo system user's guide, Release 4.0,* Chelmsford, Mass., Apr. 1982.

ARCH81 ARCHER, J. "The design and implementation of a cooperative program development environment," Ph.D. dissertation, Dep. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1981.

ARNO80 ARNOLD, C. R. C. "Screen updating and cursor movement optimization: A library package," Dep. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif, Sept. 1980.

BARA81 BARACH, D. R., TAENZER, D. H., AND WELLS, R. E. "Design of the PEN video editor display module," in *Proc ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 130–136

BAUD78 BAUDELAIRE, P C. "Draw," in *Alto user's handbook,* Xerox Palo Alto Research Center, Palo Alto, Calif, Nov. 1978, pp. 97–128.

BILO77 BILOFSKY, W. "The CRT text editor NED—Introduction and reference manual," R-2176-ARPA, Rand Corp., Santa Monica, Calif, Dec. 1977.

BBN73 BOLT BERANEK AND NEWMAN INC. *TENEX text editor and corrector manual,* Cambridge, Mass., Oct. 1973.

BOLT80 BOLT, R. A. " 'Put-that-there': Voice and gesture at the graphics interface," *Comput. Gr.* **14,** 3 (Aug. 1980), 262–270.

BOLT81 BOLT, R. A. "Gaze-orchestrated dynamic windows," *Comput. Gr.* **15,** 3 (Aug. 1981), 109–119.

BOWM81 BOWMAN, W., AND FLEGAL, B "ToolBox: A Smalltalk illustration system," *BYTE* **6,** 8 (Aug. 1981), 369–376.

BROW81 BROWN UNIVERSITY COMPUTER CENTER *User's guide to the Brown CMS editor,* Brown Univ., Providence, R. I., Apr 1981.

BURK80 BURKHART, H., AND NIEVERGELT, J. "Structure-oriented editors," Rep. 38, Eidgenossische Technische Hochschule Zurich, Institute fur Informatik, Zurich, Switzerland, May 1980.

BUSH45 BUSH, V. "As we may think," *The Atlantic Monthly* **176,** 1 (July 1945), 101–108.

CARM69 CARMODY, S., GROSS, W., NELSON, T H, RICE, D E., AND VAN DAM, A. "A hypertext editing system for the /360," in *Pertinent concepts in computer graphics,* M. Faiman and J. Nievergelt, Eds., University of Illinois Press, Urbana, Ill., 1969, pp. 291–330.

CATA79 CATANO, J. "Poetry and computers: Experimenting with the communal text," *Comput Hum.* **13** (1979), 269–275.

CHAM81 CHAMBERLIN, D. D., KING, J C., SLUTZ, D. R, TODD, S J. P., AND WADE, B. W. "JANUS· An interactive system for document composition," in *Proc ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 82–91.

COLE69 COLEMAN, M. "Text editing on a graphic display device using hand-drawn proofreader's symbols," in M. Faiman and J. Nievergelt (Eds.), *Pertinent concepts in computer graphics,* University of Illinois Press, Urbana, Ill., 1969, pp. 282–290.

COMS67 COM-SHARE, INC. *QED reference manual,* Ann Arbor, Mich , 1967

CRIS65 CRISMAN, P. A., ED. *The compatible time sharing system: A programmer's guide,* 2nd ed., M.I.T. Press, Cambridge, Mass., 1965.

DEUT67 DEUTSCH, P., AND LAMPSON, B "An online editor," *Commun. ACM* **10,** 12 (Dec. 1967), 793–799, 803.

DIGI78 DIGITAL EQUIPMENT CORPORATION *VAX-11 text editing reference manual,* Maynard, Mass., Aug. 1978.

DONZ75 DONZEAU-GOUGE, V., HUET, G., KAHN, G., LANG, B., AND LEVY, J J. "A structure oriented program editor: A first step towards computer assisted programming," Tech. Rep., IRIA-LABORIA, Rocquencourt, France, Apr 1975.

DONZ80 DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. "Programming environments based on structured editors: The MENTOR experience," Tech. Rep., INRIA, Rocquencourt, France, May 1980

EMBL81 EMBLEY, D. W., AND NAGY, G "Behavioral aspects of text editors," *Comput Surv* **13,** 1 (Mar. 1981), 33–70

ENGE63 ENGELBART, D. C. "A conceptual framework for the augmentation of man's intellect," in *Vistas in information handling,* P. Howerton, Ed., Spartan Books, Washington, D.C., 1963, pp. 1–29.

ENGE68 ENGELBART, D. C., AND ENGLISH, W. K. "A research center for augmenting human intellect," in *Proc. Fall Jt. Computer Conf.,* vol 33, AFIPS Press, Arlington, Va., fall 1968, pp. 395–410

ENGE73 ENGELBART, D. C., WATSON, R. W., AND NORTON, J. C. "The augmented knowledge workshop," in *Proc. National Computer Conf.,* vol. 42, AFIPS Press, Arlington, Va , 1973, pp. 9–21.

ENGE78 ENGELBART, D. C. "Toward integrated, evolutionary office automation systems," in *Proc. Jt Engineering Management Conf.* (Denver, Colo., Oct. 16–18, 1978), IEEE, New York, pp. 63–68.

FAJM73 FAJMAN, R. "WYLBUR: An interactive text editing and remote job entry system," *Commun. ACM* **16,** 5 (May 1973), 314–322.

FEIL80 FEILER, P. H., AND MEDINA-MORA, R. "An incremental programming environment," Tech. Rep. CMU-CS-80-126, Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Apr. 1980.

FEIN82 FEINER, S., NAGY, S., AND VAN DAM, A. "An experimental system for creating and presenting interactive graphical doc-

uments," *Trans. Gr.* **1**, 1 (Jan. 1982), 59–77.

FINS80 FINSETH, C. A. "A theory and practice of text editors," Tech. Memo. 165, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., June 1980.

FOLE82 FOLEY, J. D., AND VAN DAM, A. *Fundamentals of interactive computer graphics*, Addison-Wesley, Reading, Mass., 1982.

FRAS80 FRASER, C. W. "A generalized text editor," *Commun. ACM* **23**, 1 (Mar 1980), 27–60.

FRAS82 FRASER, C W. "Syntax-directed editing of general data structures," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 1981), ACM, New York, 1981, pp. 17–21.

FURU82 FURUTA, R., SCOFIELD, J., AND SHAW, A. "Document formatting systems· Survey, concepts, and issues," *Comput. Surv.* **14**, 3 (Sept. 1982).

GOLA79 GOLDBERG, A., AND ROBSON, D "A metaphor for user interface design," in *Proc. 12th Hawaii Int. Conf. System Sciences*, vol. 6, no. 1, 1979, pp. 148–157.

GOLA82 GOLDBERG, A. "The Smalltalk-80 system· A user guide and reference manual," Xerox Palo Alto Research Center, Palo Alto, Calif., Mar. 1, 1982; *Smalltalk80: The interactive programming environment*, Addison-Wesley, Reading, Mass., to appear in 1983.

GOLA83 GOLDBERG, A, AND ROBSON, D. *Smalltalk-80: The language and its implementation*, Addison-Wesley, Reading, Mass., to appear in 1983.

GOLC81 GOLDFARB, C. F "A generalized approach to document markup," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 68–73.

GOSL81 GOSLING, J. "A redisplay algorithm," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 123–129.

GREE80 GREENBERG, B. S. "Multics EMACS: An experiment in computer interaction," in *4th Annu Honeywell Software Conf* (Mar. 1980).

GRID82 GRID SYSTEMS CORPORATION. *Compass computer*, Mountain View, Calif., March 1982

GSPC79 GSPC, "Status report of the graphic standards planning committee," *Comput Gr.* **13**, 3 (Aug. 1979).

HABE79 HABERMANN, A. N. "An overview of the Gandalf Project," *Comput Sci. Res. Rev. 1978–79*, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979

HAMM81 HAMMER, M., ILSON, R , ANDERSON, T., GILBERT, E. J., GOOD, M., NIAMIR, B., ROSENSTEIN, L., AND SCHOICHET, S. "The implementation of Etude, an integrated and interactive document production system," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 137–146.

HANS71 HANSEN, W. J. "Creation of hierarchic text with a computer display," Rep. ANL7818, Argonne National Laboratory, Argonne, Ill., July 1971.

HERO80 HEROT, C., CARLING, R., FRIEDELL, M., AND KRAMLICH, D. "A prototype spatial data management system," *Comput. Gr.* **14**, 3 (July 1980), 63–70.

IBM67 INTERNATIONAL BUSINESS MACHINES *Magnetic tape selectric typewriter*, New York, 1967

IBM69 INTERNATIONAL BUSINESS MACHINES *A conversational, context-directed editor*, Cambridge, Mass., July 1969.

IBM70 INTERNATIONAL BUSINESS MACHINES *System/360 administrative terminal system*, 1970.

IBM80 INTERNATIONAL BUSINESS MACHINES *IBM virtual machine/system product: System product editor user's guide*, White Plains, N. Y., July 1980.

IMAG81 IMAGE DATA PRODUCTS, LTD. *The image data tablet*, Bristol, England, 1981.

IRON72 IRONS, E. T., AND DJORUP, F. M "A CRT editing system," *Commun. ACM* **15**, 1 (Jan. 1972), 16–20.

JOHN75 JOHNSON, S. "YACC—Yet another compiler-compiler," Bell Laboratories, Murray Hill, N. J., 1975.

JOY80a JOY, W., AND HORTON, M. "Ex reference manual—Version 3.1," Dep. of Electrical Engineering and Computer Science, Univ of California, Berkeley, Calif., Sept. 16, 1980.

JOY80b JOY, W., AND HORTON, M. "An introduction to display editing with Vi," Dep. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif., Sept. 16, 1980.

JOY81 JOY, W., AND HORTON, M. "TERMCAP," in *UNIX programmer's manual*, 7th ed., Berkeley Release 4.1, Dep. of Electrical Engineering and Computer Science, Univ. of California, Berkeley Calif, June, 1981.

KAY77 KAY, A., AND GOLDBERG, A. "Personal dynamic media," *Computer* **10**, 3 (Mar. 1977), 31–43.

KELL77 KELLY, J. "A guide to NED· A new online computer editor," Rep. R-2000-ARPA, Rand Corp., Santa Monica, Calif, July 1977.

KERN78a KERNIGHAN, B. W. "A tutorial introduction to the UNIX editor," Bell Laboratories, Murray Hill, N. J., Sept. 28, 1978.

KERN78b KERNIGHAN, B. W. "Advanced editing on UNIX," Bell Laboratories, Murray Hill, N. J , Aug. 4, 1978.

KERN78c KERNIGHAN, B. W., AND RITCHIE, D. M. *The C programming language*, Prentice-Hall, Englewood Cliffs, N. J., 1978.

KERN82 KERNIGHAN, B. W., AND LESK, M. E. In *Document preparation systems*, J. Nievergelt, G. Coray, J. Nicoud, and A. Shaw, Eds., North-Holland Publ., Amsterdam and New York, 1982.

KNUT79 KNUTH, D. E. *TEX and metafont: New directions in typesetting*, Digital Press, Bedford, Mass., Dec. 1979.

LAMP78 LAMPSON, B. W. "Bravo manual," in *Alto user's handbook*, Xerox Palo Alto Research Center, Palo Alto, Calif., Nov 1978, pp. 31–62.

LANT79 LANTZ, K., AND RASHID, R. "Virtual terminal management in a multiple process environment,'j' in *Proc. 7th Symp. Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12, 1979), ACM, New York, 1979, pp. 86–95.

LANT80 LANTZ, K "Uniform interfaces for distributed systems," Computer Science Dep., Univ. of Rochester, Rochester, N. Y, May 1980.

LESK81 LESK, M. "Another view," *Datamation* **27**, 12 (Nov. 1981), 146.

LRG76 LEARNING RESEARCH GROUP "Personal dynamic media," Tech. Rep. SSL-76-1, Xerox Palo Alto Research Center, Palo Alto, Calif., Mar. 1976.

MAXE81 MAXEMCHUCK, N. F., AND WILDER, H. A. "Virtual editing: I. The concept," in *Proc. 2nd Int. Workshop on Office Information Systems* (Couvent Royal de St. Maximin, Oct. 13–15), Elsevier North-Holland, New York, 1982, pp. 161–172.

MCCA67 MCCARTHY, J., DOW, B., FELDMAN, G., AND ALLEN, J. "THOR—A display based timesharing system," in *Proc. Spring Jt. Computer Conf* vol. 30, AFIPS Press, Arlington, Va., Spring 1967, pp. 623–633.

MEDI81 MEDINA-MORA, R., AND NOTKIN, D. S. "ALOE users' and implementor's guide," Tech. Rep., Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa , Nov. 1981.

MEYR81 MEYROWITZ, N., AND MOSER, M. "BRUWIN: An adaptable design strategy for window manager/virtual terminal systems," in *Proc. 8th Symp. Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16, 1981), ACM, New York, 1981, pp. 180–189.

MICR81 MICROPRO *WordStar user's guide*, MicroPro International Corp., San Rafael, Calif., 1981.

MONT82 MONTGOMERY, E. B. "Bringing manual input into the 20th Century: New keyboard concepts," *IEEE Comput.* **15**, 3 (Mar. 1982), 11–18.

MORR81 MORRIS, J. M., AND SCHWARTZ, M. D. "The design of a language-directed editor for block-structured languages," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 28–33.

NELS67 NELSON, T. H. "Getting it out of our system," in *Information retrieval: A critical review*, G. Schecter, Ed., Thompson Book Co., Washington, D. C., 1967, pp. 191–210.

NELS74 NELSON, T. H. *Computer lib/dream machines*, Hugo's Book Service, Chicago, Ill., 1974.

NELS81 NELSON, T. H. *Literary machines*, T. H. Nelson, Swarthmore, Pa., 1981.

NEWM78 NEWMAN, W. M. "Markup," in *Alto user's handbook*, Xerox Palo Alto Research Center, Palo Alto, Calif., Nov. 1978, pp. 85–96.

NEWM79 NEWMAN, W., AND SPROULL, R. *Principles of interactive computer graphics*, McGraw-Hill, New York, 1979

NORM81 NORMAN, D A. "The trouble with UNIX," *Datamation* **27**, 12 (Nov. 1981), 139–150.

NOTK79 NOTKIN, D S., AND HABERMANN, A. N. "Software development environment issues as related to Ada," Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979.

PRUS79 PRUSKY, J. N. *FRESS resource manual*, Dep. of Computer Science, Brown Univ., Providence, R. I., 1979.

REID80a REID, B K. "A high-level approach to computer document formatting," in *Proc 7th Annu. ACM Symp. Programming Languages* (Jan. 1980), ACM, New York, 1980, pp. 24–30.

REID80b REID, B. K. "Scribe: A document specification language and its compiler," Ph.D. dissertation, Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1980.

REID80c REID, B. K., AND WALKER, J. H. *Scribe user's manual*, 3rd ed., Unilogic, Ltd., Pittsburgh, Pa., 1980.

REIS81 REISS, S. P., LUSTIG, M., AND MEDVENE, L. *bb user's guide*, Dep. of Computer Science, Brown Univ., Providence, R. I., 1981.

RICE71 RICE, D. E., AND VAN DAM, A "An introduction to information structures and paging considerations for on-line text editing systems," in *Advances in information systems science*, J. Tou, Ed., Plenum Press, New York, 1971, pp. 93–159

ROBG79 ROBERTSON, G , MCCRACKEN, D., AND NEWELL, A. "The ZOG approach to man-machine communication," Tech. Rep. CMU-CS-79-148, Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Oct. 1979.

SEYJ81 SEYBOLD, J. "Xerox's 'Star,'" *Seybold Rep.* **10**, 16 (Apr. 27, 1981).

SEYP79 SEYBOLD, P. B. "The CPT 8000 and 6000 word processing systems," *Seybold Rep. Word Process.* **2**, 1 (Feb. 1979), 1–16.

SMIT82 SMITH, D. C., IRBY, C., KIMBALL, R., VERPLANK, B., AND HARSLEM, E. "De-

signing the Star user interface," *BYTE* 7, 4 (Apr. 1982), 242-282.

STAL80 STALLMAN, R. M. "EMACS manual for TWENEX users," AI Memo. 556, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., Aug. 17, 1980.

STAL81 STALLMAN, R. M. "EMACS: The extensible, customizable self-documenting display editor," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 147-156.

SUTH63 SUTHERLAND, I. E. "THOR—A display based timesharing system," in *Proc. Spring Jt. Computer Conf.*, vol. 23, Spartan, Baltimore, 1963, p. 329.

SWIN74 SWINEHART, D. "Copilot: A Multiple process approach to interactive programming systems," Ph.D. dissertation, Stanford Artificial Intelligence Laboratory Memo. AIM-230, Stanford Univ., Palo Alto, Calif., July 1974.

SYMB81 SYMBOLICS, INC. *Symbolics software*, Woodland Hills, Calif., 1981.

TEIT81a TEITELBAUM, T., AND REPS, T. "The Cornell program synthesizer· A syntax-directed programming environment," *Commun. ACM* **24**, 9 (Sept. 1981), 563-573.

TEIT81b TEITELBAUM, T., REPS, T, AND HORWITZ, S. "The why and wherefore of the Cornell program synthesizer," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 1981), ACM, New York, 1981, pp. 8-16.

TEIW77 TEITELMAN, W. "A display oriented programmer's assistant," Rep. CSL-77-3, Xerox Palo Alto Research Center, Palo Alto, Calif., Mar. 1977.

TESL81 TESLER, L. "The Smalltalk environment," *BYTE*, **6**, 8 (Aug. 1981), 90-147.

TILB76 TILBROOK, D. "A newspaper page layout system," M. Sc. thesis, Dep. of Computer Science, University of Toronto, Toronto, Canada, 1976.

TOLL65 TOLLIVER, B. "TVEDIT," Stanford Time-Sharing Memo. No. 32, Dep. of Computer Science, Stanford Univ., Palo Alto, Calif., 1965.

VAND71a VAN DAM, A. *FRESS (file retrieval and editing system)*, Text Systems, Barrington, R. I., July 1971.

VAND71b VAN DAM, A., AND RICE, D. E. "On-line text editing: A survey," *Comput. Surv.* **3**, 3 (Sept. 1971), 93-114.

VIP69 VIP SYSTEMS *VIPcom user's guide*, Washington, D. C., 1968.

WALK81a WALKER, J. H. Personal communication, July 1981

WALK81b WALKER, J. H. "The document editor: A supporting environment for preparing technical documents," in *Proc ACM SIGPLAN/SIGOA Symp. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 44-50

WILC76 WILCOX, T. R., DAVIS, A. M., AND TINDALL, M. H. "The design and implementation of a table-driven, interactive diagnostic programming system," *Commun. ACM* **19**, 11 (Nov. 1976), 609-616

WILD82 WILDER, H. A., AND MAXEMCHUCK, N. F. "Virtual editing: II. The user interface," in *Proc. SIGOA Conf. Office Automation Systems* (Philadelphia, Pa., June 21-23, 1982), ACM, New York, 1982, pp. 41-46.

WOOD81 WOODS, S. R. "Z—The 95 percent program editor," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 1-7.

XERO82 XEROX CORPORATION. *8010 Star information system reference guide*, Dallas, Tex., 1982.

BIBLIOGRAPHY

ALBE79 ALBERGA, C. N., BROWN, A. L., LEEMAN, G. B. JR., MIKELSONS, M., AND WEGMAN, M. N. "A program development tool," Tech. Rep. RC 7859, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Sept. 1979.

BEAC81 BEACH, R. J., BEATTY, J. C., BOOTH, K. S., AND WHITE, A. R. "Documentation graphics at the University of Waterloo," in *Int. Conf. Research Trends in Document Preparation Systems* (Lausanne, Switzerland, Feb. 27-28, 1981), Swiss Institutes of Technology, Lausanne and Zurich, pp. 123-125.

BORK79 BORK, A. "Textual taxonomy," Educational Technology Center, Physics Dep., Univ. of California, Irvine, Calif., Oct. 4, 1979.

BORK81 BORKIN, S. A., AND PRAGER, J. M. "Some issues in the design of an editor-formatter for structured documents," Tech. Rep., IBM Cambridge Scientific Center, Cambridge, Mass., Sept. 1981.

BURK81 BURKHART, H., AND STELOVSKY, J. "Towards an integration of editors," in *Int. Conf. Research Trends in Document Preparation Systems* (Lausanne, Switzerland, Feb. 27-28, 1981), Swiss Institutes of Technology, Lausanne and Zurich, pp. 9-11.

BUSH67 BUSH, V. "Memex revisted," in *Science Is Not Enough*, V. Bush, Ed., William Morrow, 1967, pp. 75-101.

CARD76 CARD, S. K., MORAN, P., AND NEWELL, A. "The manuscript editing task: A routine cognitive skill," Rep. SSL-76-8, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Calif., Dec. 1976.

CARD78a CARD, S. K. "Studies in the psychology of computer text editing," Rep SSL-78-1, Xerox Palo Alto Research Center, Palo Alto, Calif., Aug. 1978

CARD78b CARD, S. K., ENGLISH, W. K., AND BURR,

B. J. "Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT," *Ergonomics* **21** (1978), 601–613.

CARD80 CARD, S. K., MORAN, P. T., AND NEWELL, A. "The keystroke-level model for user performance time with interactive systems," *Commun ACM* **23**, 7 (July 1980), 396–410.

CHER81 CHERRY, L. "Computer aids for writers," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore, June 8–10, 1981), ACM, New York, 1981, pp. 61–67.

COUL76 COULOURIS, G. F., DURHAM, I., HUTCHINSON, J. R., PATEL, M. H., REEVES, T, AND WINDERBANK, D. G. "The design and implementation of an interactive document editor," *Softw. Pract. Exper.* **6**, 2 (May 1976), 271–279.

DZID78 DZIDA, W., HERDA, S., AND ITZFELDT, W. D. "User-perceived quality of interactive systems," *IEEE Trans. Softw Eng.* **SE-4**, 4 (July 1978), 270–276.

ELLI80 ELLIS, C., AND NUTT, G "Office information systems and computer science," *Comput. Surveys* **12**, 1 (Mar. 1980), 27–60.

EMBL78 EMBLEY, D W., LAN, M. T., LEINBAUGH, D. W., AND NAGY, G. "A procedure for predicting program editor performance from the user's point of view," *Int J. Man-Mach. Stud.* **10**, 6 (Nov. 1978), 639–650.

ENGL67 ENGLISH, W. K., ENGELBART, D. C., AND BERMAN, M. L. "Display-selection techniques for text manipulation," *IEEE Trans. Hum Factors Electron.* **HFE-8**, 1 (Mar 1967), 5–15.

FEIN81a FEINER, S, NAGY, S., AND VAN DAM, A. "An integrated system for creating and presenting complex computer-based documents," *Comput. Gr.* **15**, 3 (Aug. 1981), 181–189.

FEIN81b FEINER, S., NAGY, S., AND VAN DAM, A. "Online documents combining pictures and text," in *Proc. Int. Conf. Research and Trends in Document Preparation Systems* (Lausanne, Switzerland, Feb 27–28, 1981), Swiss Institutes of Technology, Lausanne and Zurich, pp. 1–4.

FINS82 FINSETH, C. A. "Managing words· What capabilities should you have with a text editor?," *BYTE* **7**, 4 (Apr. 1982), 242–282.

FRAS79 FRASER, C. W. "A compact, portable CRT-based text editor," *Softw. Pract. Exper.* **9**, 2 (Feb 1979), 121–125.

FREI78 FREI, H. P., WELLER, D. L., AND WILLIAMS, R "A graphics-based programming-support system," *Comput. Gr.* **12**, 3 (Aug. 1978), 43–49

GOLA80 GOLDBERG, A., AND ROBSON, D. "Sharing problems: Personal computers as interpersonal tools," Keynote Address at the SIGSMALL/PC Symp. (Palo Alto, Calif., Sept. 1980), Xerox Palo Alto Research Center, Palo Alto, Calif., 1980.

GOOM81 GOOD, M. "Etude and the folklore of user interface design," in *Proc. ACM SIGPLAN/SIGOA Symp. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 34–43.

GOON75 GOODWIN, N. C. "Cursor positioning on an electronic display using lightpen, lightgun or keyboard for three basic tasks," *Hum. Factors* **17**, 3 (June 1975), 289–295.

HANS68 HANSEN, W. J. "User engineering principles for interactive systems," in *Proc. fall Jt. Computer Conf.*, vol. 39, AFIPS Press, Arlington, Va., fall 1968, pp. 395–410.

HAYE81 HAYES, P., BALL, E., AND REDDY, R. "Breaking the man-machine communication barrier," *Computer* **14**, 3 (Mar. 1981), 19–30.

HAZE80 HAZEL, P. "Development of the ZED text editor," *Softw Pract. Exper.* **10**, 1 (Jan. 1980), 57–76

JONG82 JONG, S. "Designing a text editor? The user comes first," *BYTE* **7**, 4 (Apr 1982), 284–300.

KERN75 KERNIGHAN, B. W., AND CHERRY, L. L. "A system for typesetting mathematics," *Commun. ACM* **18**, 3 (Mar. 1975), 182–193.

KERN81 KERNIGHAN, B. W. "PIC—A language for typesetting graphics," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 92–98.

LEDE80 LEDERMAN, A. "An abstracted bibliography on programming environments," Dep. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass, June 1980.

LEDG80 LEDGARD, H. "The natural language of interactive systems," *Commun. ACM* **23**, 10 (Oct. 1980), 556–563.

LESK76 LESK, M. "TBL—A program to format tables," Tech. Rep. 49, Bell Laboratories, Murray Hill, N. J., 1976.

MACD80 MACDONALD, N. H, FRASE, L. T., AND KEENAN, S. A. "Writer's workbench: Computer programs for text editing and assessment," Bell Laboratories, Piscataway, N. J., 1980.

MACL77 MACLEOD, I. A. "Design and implementation of a display-oriented text editor," *Softw. Pract. Exper.* **7**, 6 (Nov. 1977), 771–778.

MIKE81 MIKELSONS, M. "Prettyprinting in an interactive programming environment," in *Proc ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8–10, 1981), ACM, New York, 1981, pp. 108–116.

MUKH80 MUKHOPADHYAY, A. "A proposal for a hardware text processor," in *Papers 5th Workshop on Computer Architecture for*

Non-Numeric Processing (Pacific Grove, Calif., Mar. 11-14, 1980), ACM, New York, pp. 57-61

NASS73 NASSI, I., AND SHNEIDERMAN, B. "Flowchart techniques for structured programming," *ACM SIGPLAN Not.* **8,** 8 (Aug. 1973), 12-26.

NBI81 NBI, INC. *System 3000 operator's guide,* Boulder, Colo., Mar. 1981.

OSSA76 OSSANNA, J. "NROFF/TROFF user's manual," Tech. Rep. 54, Bell Laboratories, Murray Hill, N. J., 1976.

PETE80 PETERSON, J. L. "Computer programs for detecting and correcting spelling errors," *Commun. ACM* **23,** 12 (Dec 1980).

REID81 REID, B. K., AND HANSON, D. "An annotated bibliography of background material on text manipulation," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 157-160.

ROBT79 ROBERTS, T. "Evaluation of computer text editors," Rep. SSL-79-9, Xerox Palo Alto Research Center, Palo Alto, Calif., Nov. 1979.

SAND78 SANDEWALL, E. "Programming in an interactive environment: The LISP experience," *ACM Comput. Surv* **10,** 1 (Mar. 1978), 35-71.

SEYP78 SEYBOLD, P B. "Tymshare's augment. Heralding a new era," *Seybold Rep. Word Process.* **1,** 9 (Oct. 1978), 1-16.

SNEE78 SNEERINGER, J. "User-interface design for text editing: A case study," *Softw. Pract. Exper.* **8,** 5 (Sept.-Oct. 1978), 544-557.

STRO81 STROMFORS, O., AND JONESJO, L. "The implementation and experiences of a structure-oriented text editor," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 22-27.

SUFR81 SUFRIN, B. "Formal specification of a display editor," Tech. Monogr. PRG-21, Programming Research Group, Computing Laboratory, Oxford University, June 1981.

TESL79 TESLER, L. "Home text editing: A tutorial," Xerox Palo Alto Research Center, Palo Alto, Calif., 1979.

THIM78 THIMBLEBY, H. "Character oriented text editing," Computer Systems Laboratory, Queen Mary College, London University, London, England, Nov. 1978.

TURB81 TURBA, T. N. "Checking for spelling and typographical errors in computer-based text," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 51-60.

VANW81 VAN WYK, C. J. "A graphics typesetting language," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, New York, 1981, pp. 99-107.

WHIT81 WHITE, A. R. "Pic—A C-based illustration language," Dep. of Computer Science, Univ of Waterloo, Waterloo, Ontario, Canada, 1981.

Received August 1981; final revision accepted June 1982.